Computer Architecture

Lecture 07 – Address Translation & Virtual Mem

Tian Xia
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University
http://gr.xjtu.edu.cn/web/pengjuren

Any problem in computer science can be solved with another layer of indirection.

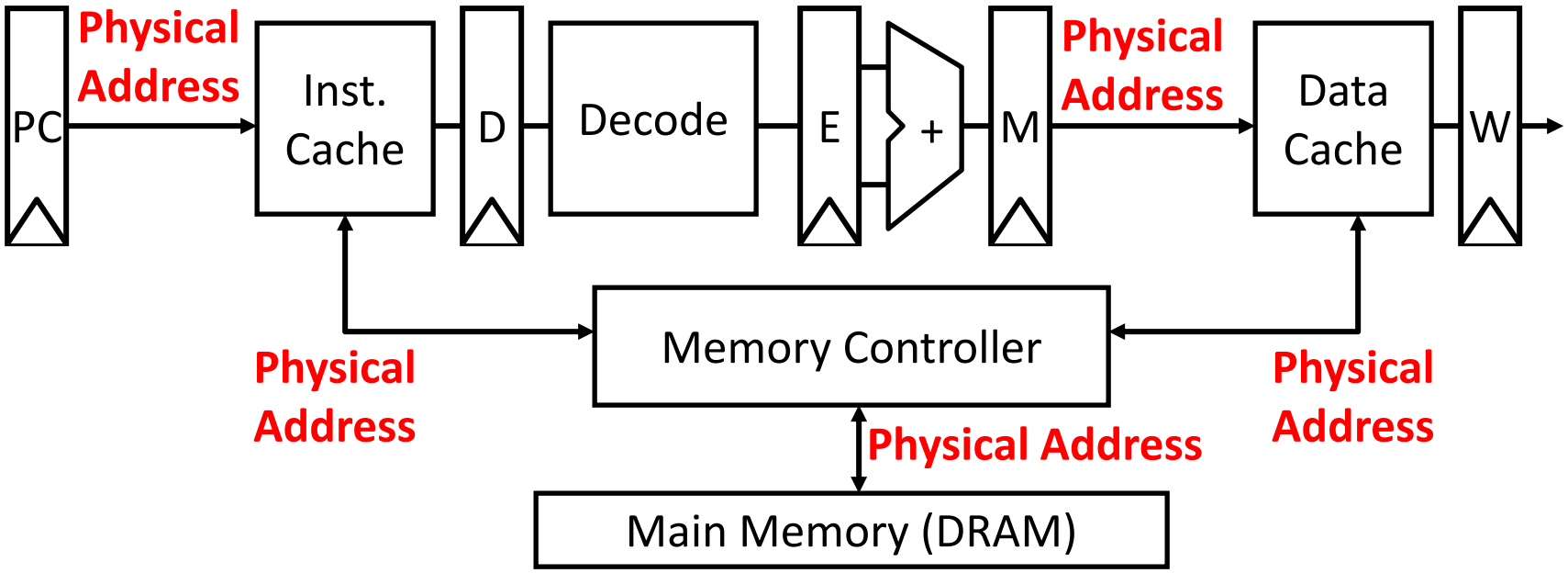
--David Wheeler

British Computer Scientist

(1927--2004)

- Fellow of the Royal Society (1981)
- Computer Pioneer Award (1985)
- Fellow, Computer History Museum (2003)

Bare Machine



In a bare machine, the only kind of address is a **physical address**, corresponding to address lines of actual hardware memory.

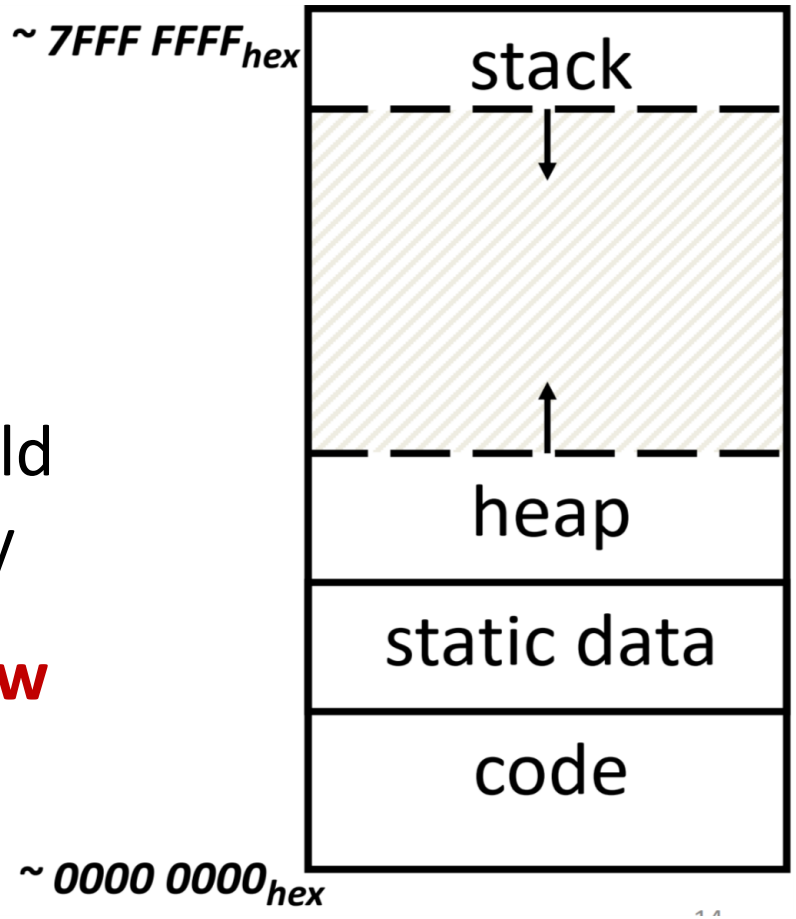
Managing Memory in Bare Machines

- Early machines only ran **one program at a time**, with this program having unrestricted access to all memory and all I/O devices
  - This simple memory management model was also used in turn by the first minicomputer and first microcomputer systems
- Subroutine **libraries** became popular, were written in **location-independent** form
  - Different programs use different combination of routines
- To run program on bare machines, use **linker** and **loader** program to relocate library modules to actual locations in physical memory

Why do we need Virtual Address Space ?

Reason 1: Simplifying Memory for Apps

- Programmers should see the **straight-forward memory layout** as we assume →
- User-space applications should **think** they own all of memory
- So we give them a **virtual view** of memory

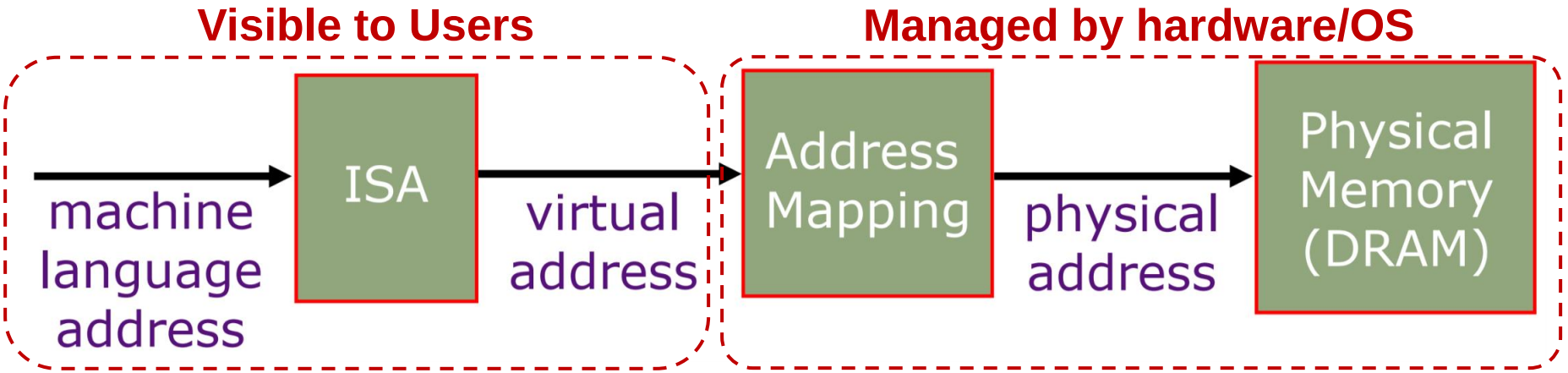


Why do we need Virtual Address Space ?

Reason 2: Protection Between Processes

- With a bare system, addresses issued with loads/stores are real **physical** addresses
- This means any program can issue any address, therefore can access **any part of memory**, even areas which it doesn't own
  – Ex: The OS data structures
- We should send all addresses through a mechanism that the **OS controls**, before they make it out to DRAM
  - ***a translation and protection mechanism***

Names for Virtual Memory Locations



- Machine Language address
  - As specified in machine code
- **Virtual Address (VA)**
  - ISA specifies translation of machine code address into virtual address of program variable (sometime called **effective/logical address**)
- **Physical Address (PA)**
  - Operating System specifies **mapping/translation** of virtual address into name for a physical memory location

The Common Denominator: Address Translation

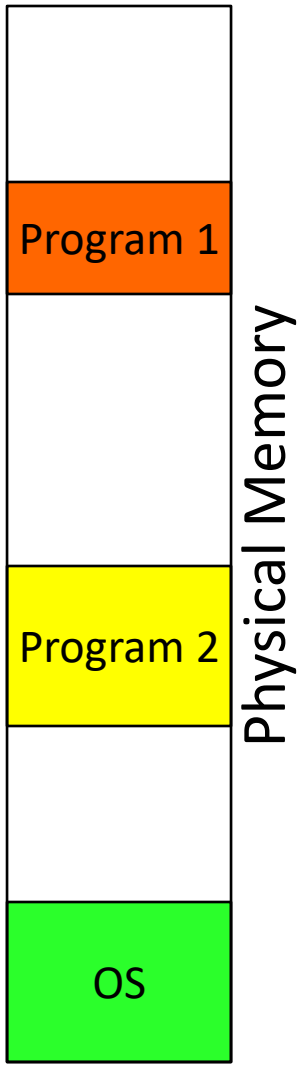
- Large, private, and uniform abstraction achieved through address translation
  - User process operates on virtual address (**VA**)
  - HW translates **VA** to physical address (**PA**) on every **memory reference**
- Through address translation
  - **Control** which **physical locations** (DRAM and/or disk) can be referred to by a process
  - **Dynamic allocation** and relocation of physical backing store (where in DRAM and/or disk)
- Address translation HW and policies **controlled** by the **OS** and **protected** from user

Simple Base and Bound Translation

- Problems of the bare memory system
  - Each process limited to a non-overlapping contiguous physical memory region (space doesn't start from address 0…)
  - Everything must fit in the region
- Location-independent programs
  - Ease programming and storage management

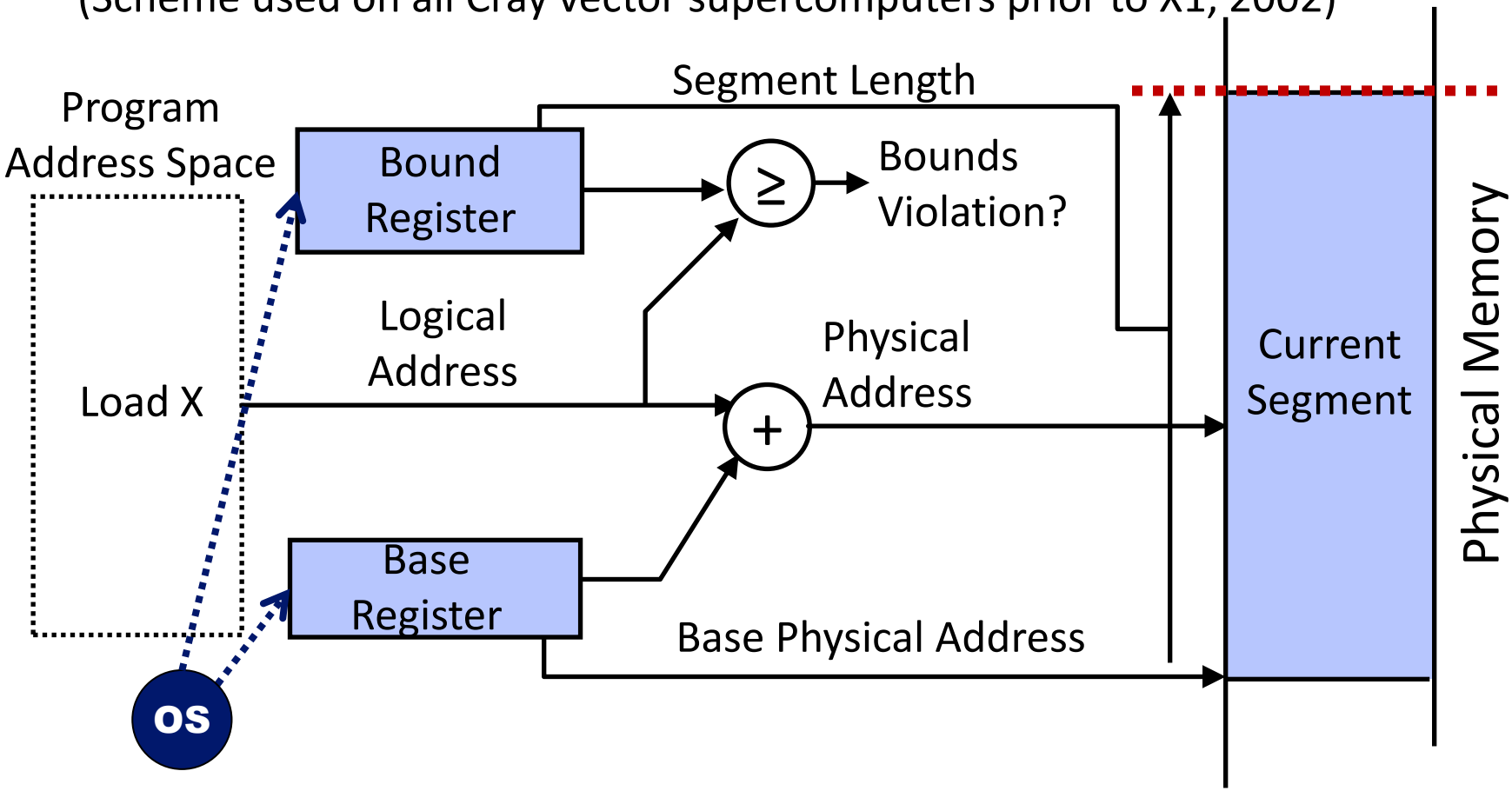
  → need for a *Base* register
- Protection
  - Independent programs should not affect each other inadvertently

  → need for a *Bound* register
- Multiprogramming drives requirement for **resident supervisor software** (e.g. OS) to manage context switches between multiple programs



Simple Base and Bound Translation

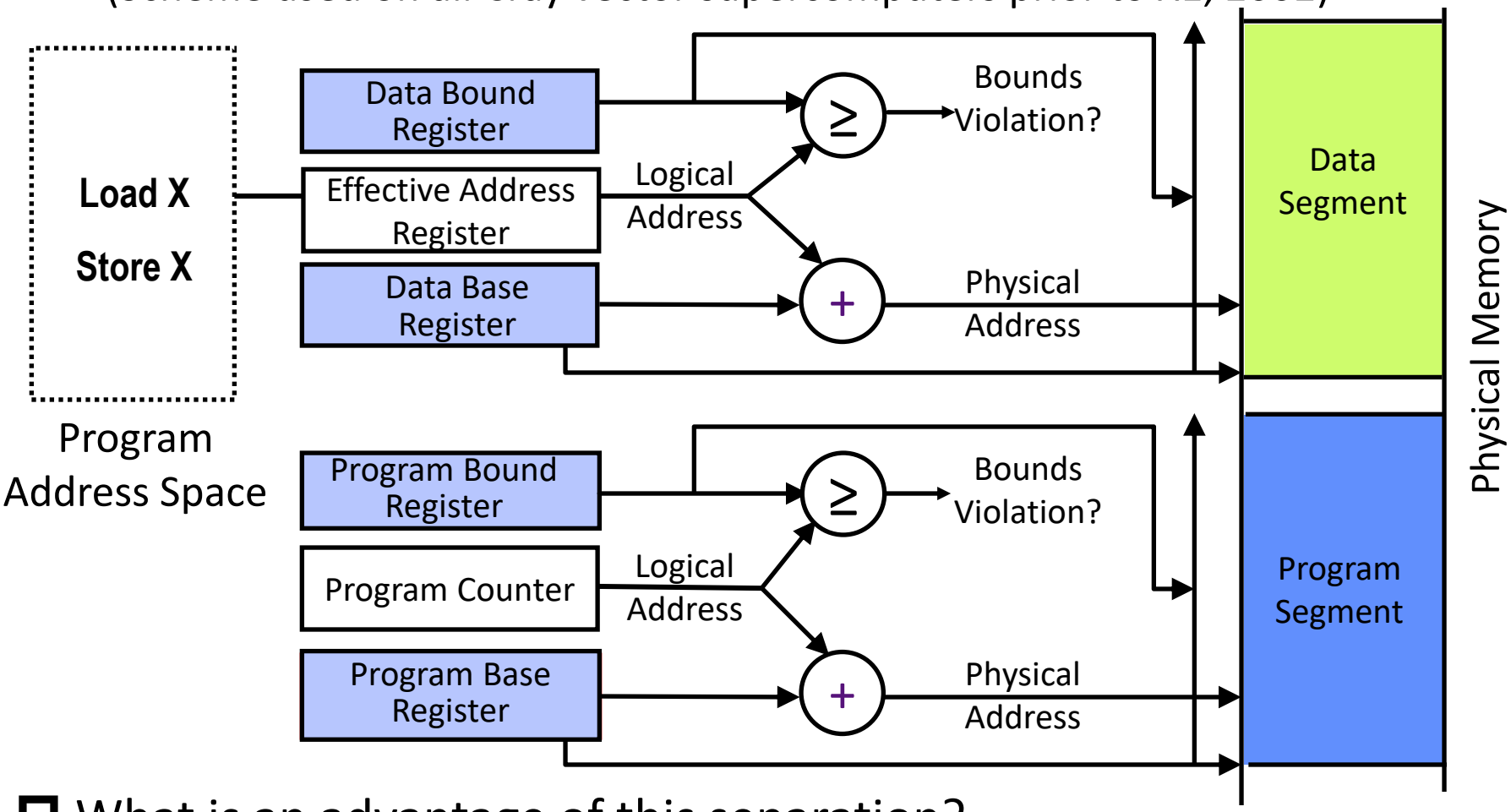
(Scheme used on all Cray vector supercomputers prior to X1, 2002)



- **Base** (Staring address) and **Bound** (size of region) registers are visible/accessible only when processor is running in the ***supervisor mode*** *(privileged control registers)*
- **OS** switches bound/base register pair for different programs

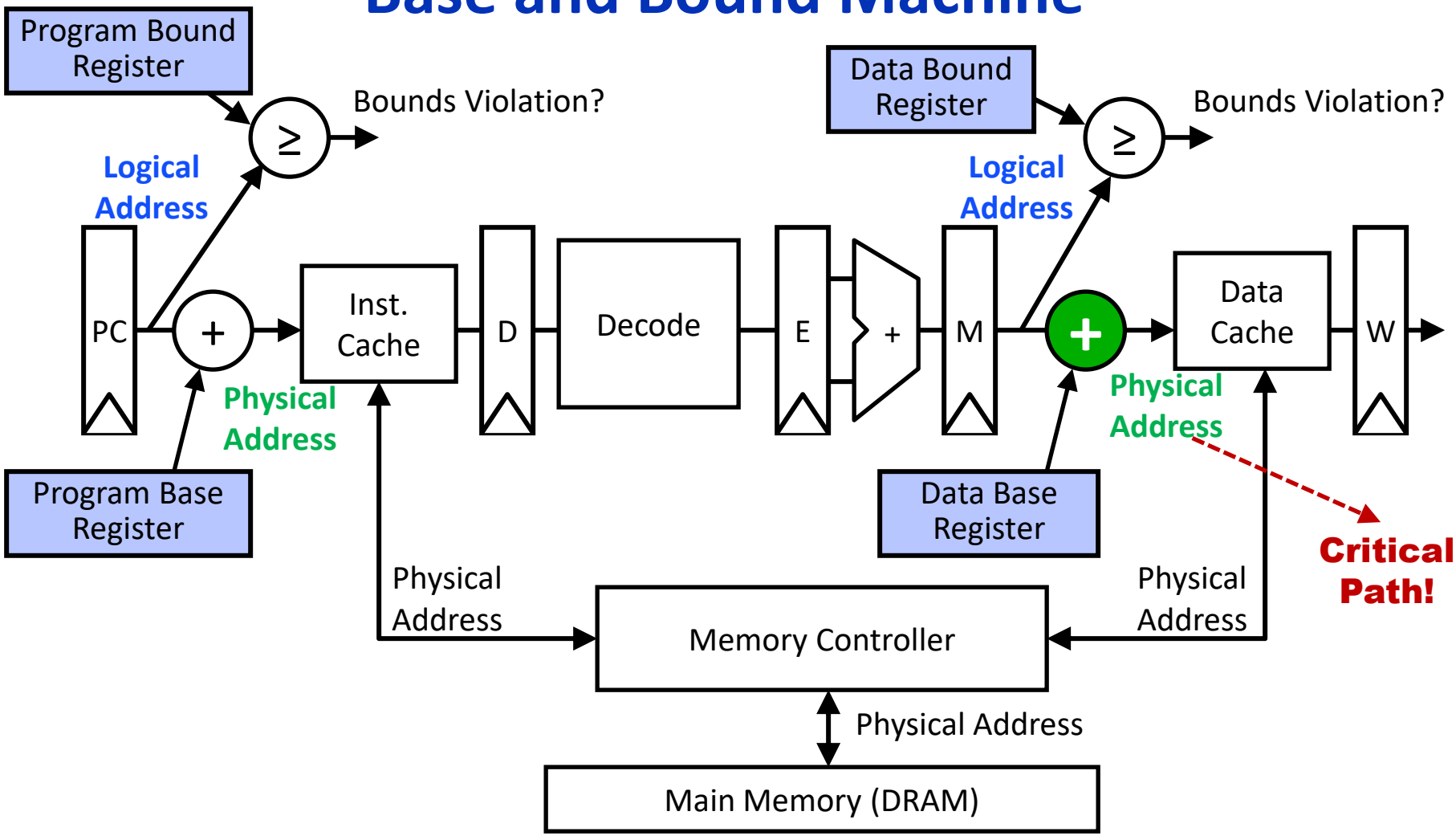
Separate Areas for Program and Data

(Scheme used on all Cray vector supercomputers prior to X1, 2002)



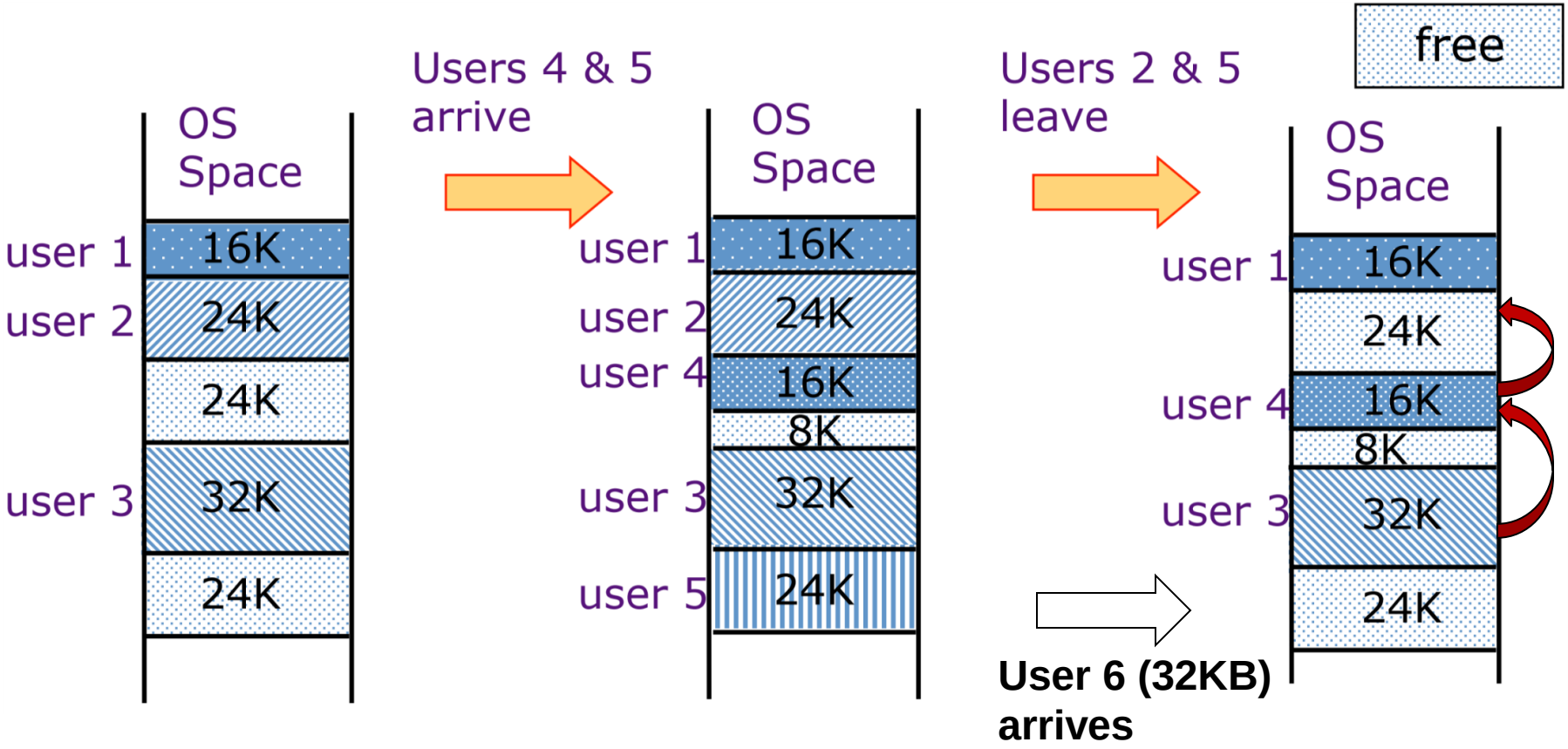
- What is an advantage of this separation?
  - **Shared program** segment with **independent data** segment
  - **Very fast** translation
- Each program has **only one** data segment

Base and Bound Machine



Can fold **addition of base register** into (register + immediate) **address calculation** using a carry-save adder (sums three numbers with only a few gate delays more than adding two numbers)

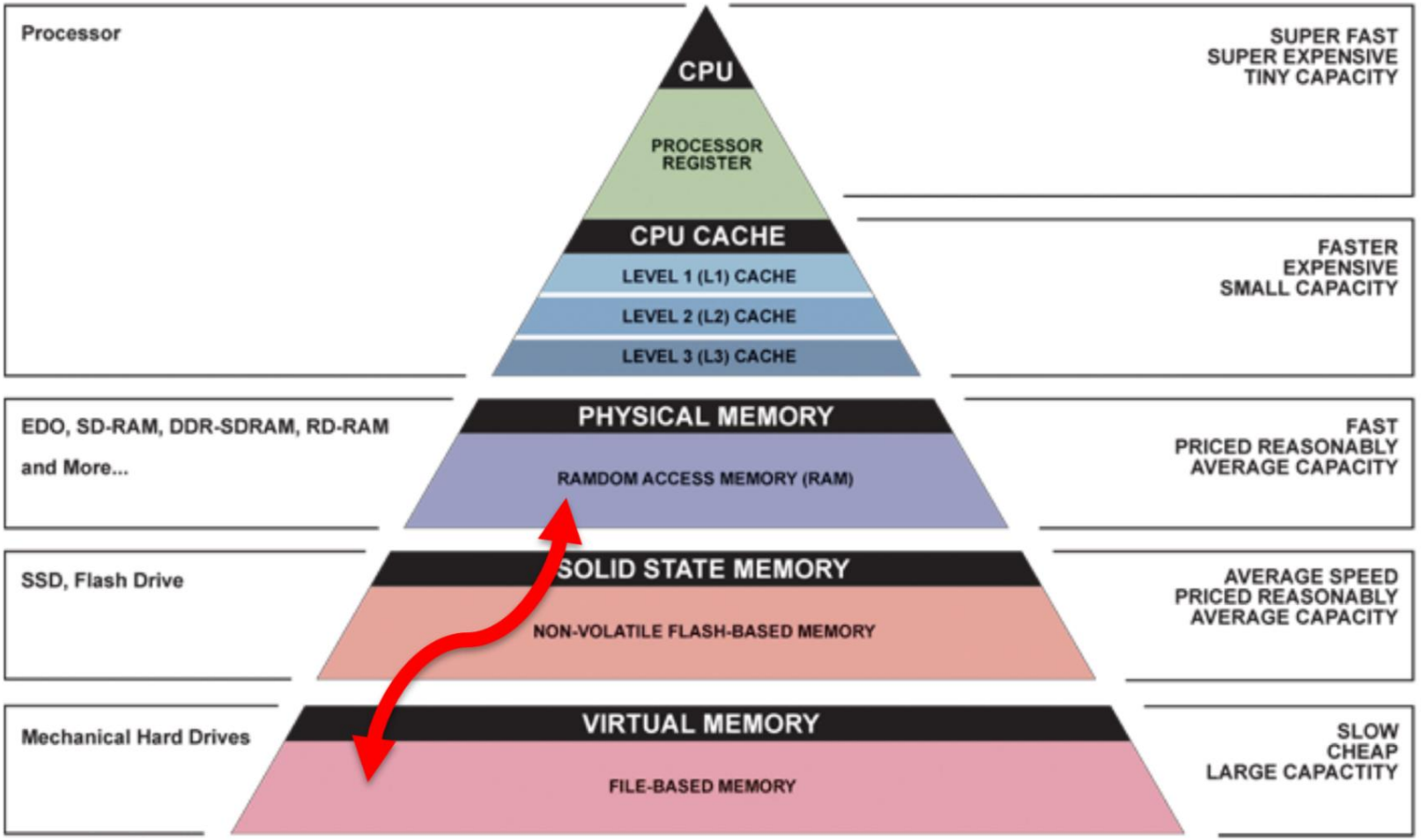
External Fragmentation with Segments



As users come and go, the storage is "fragmented“(external)

- **Plan ahead** to avoid bubbles
- Programs are **moved around** to compact the storage

Adding Disks to Hierarchy



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

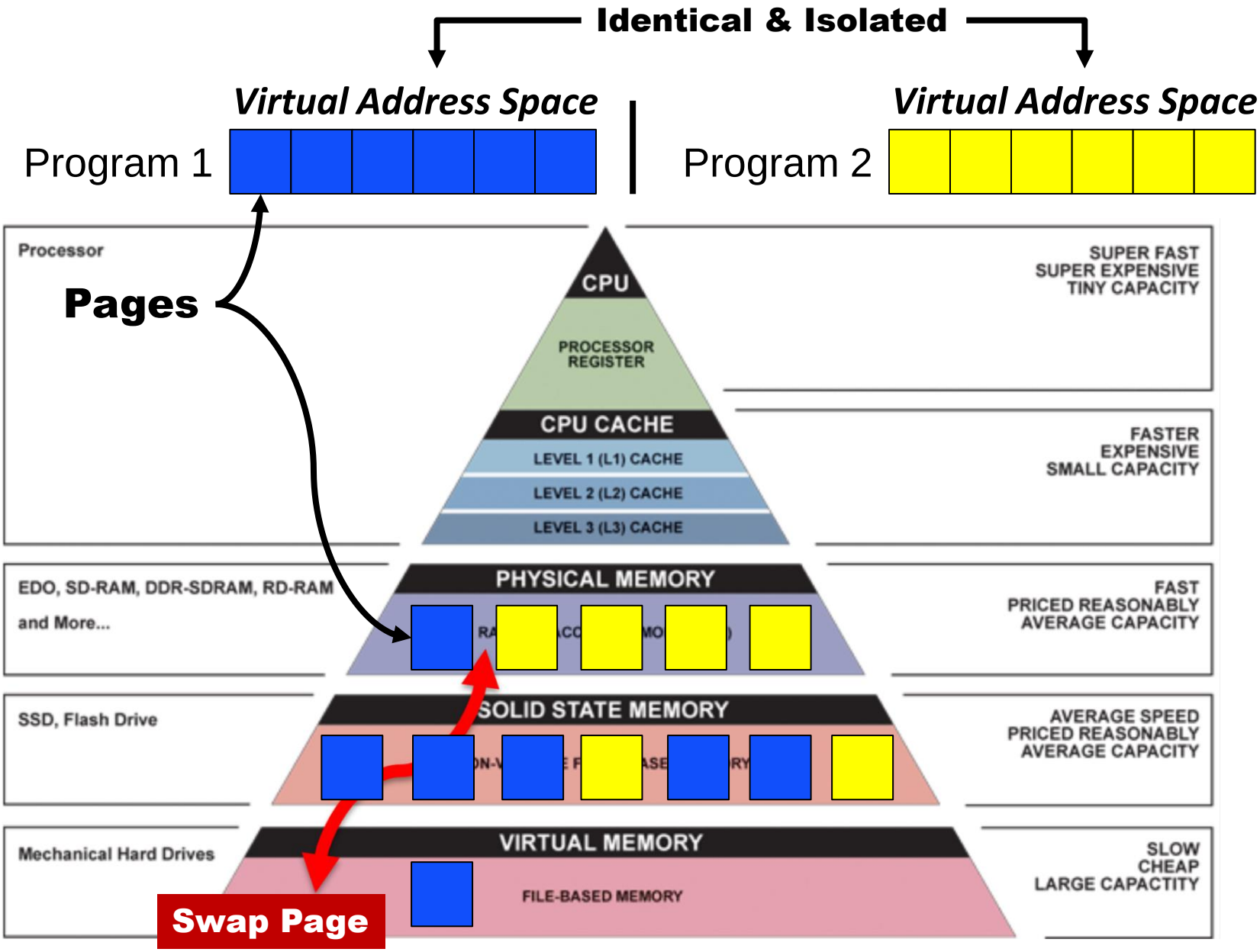
Need to devise a mechanism to "**Connect**" memory and disk in the memory hierarchy.

Two Ingredients to Modern Virtual Memory

In a multi-tasking system, virtual memory supports the **illusion** of a **large**, **private**, and **uniform** memory space to each process

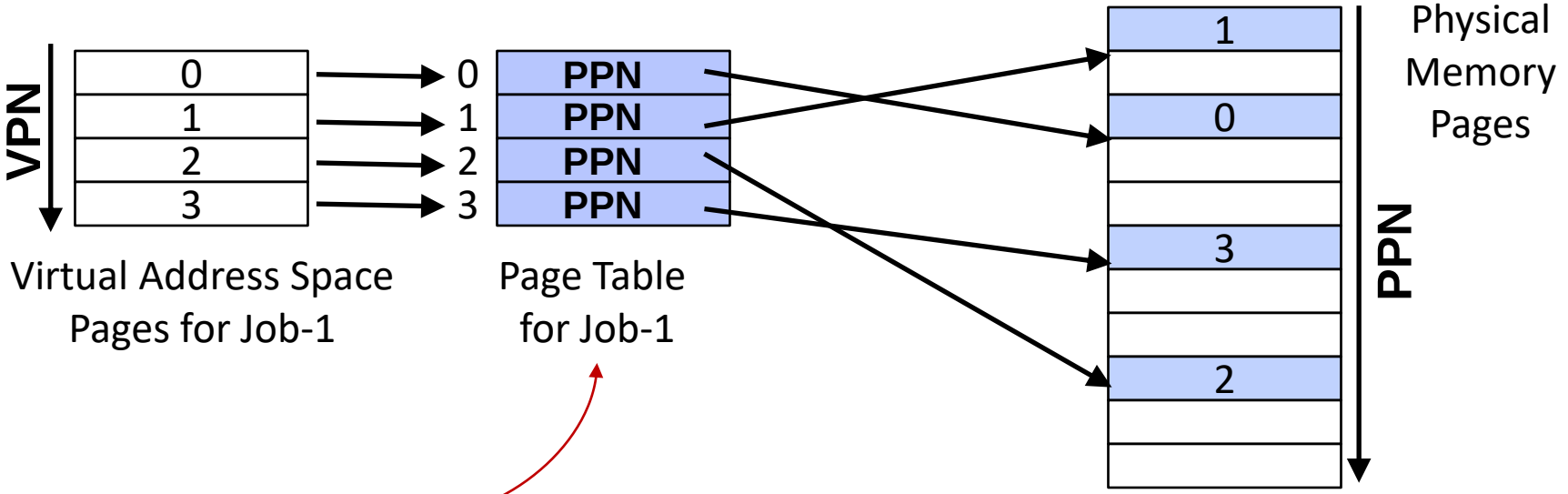
- Ingredient A: independence and protection
  - location-independent programs
  - each process sees a large, contiguous address space without holes ***(for programming convenience)***
  - each process's memory is private, i.e., protected from access by other processes ***(for sharing and protection, Readable? Writeable? Executable?)***
- Ingredient B: demand paging ***(for hierarchy and efficiency)***
  - **Dynamic allocation** and relocation of **physical pages** (where in DRAM and/or swap disk)
  - **Capacity** of secondary storage (swap space on disk)
  - **Speed** of primary storage (DRAM)

Modern Paged Virtual Memory System

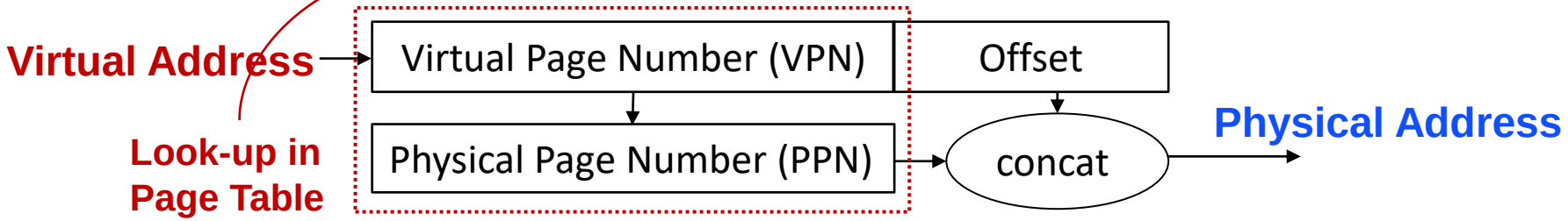


Paged Virtual Memory System (How)

- **Fixed-sized pages (mostly 4KB)** in virtual address space are mapped to physical address using **Page Table (PT)**



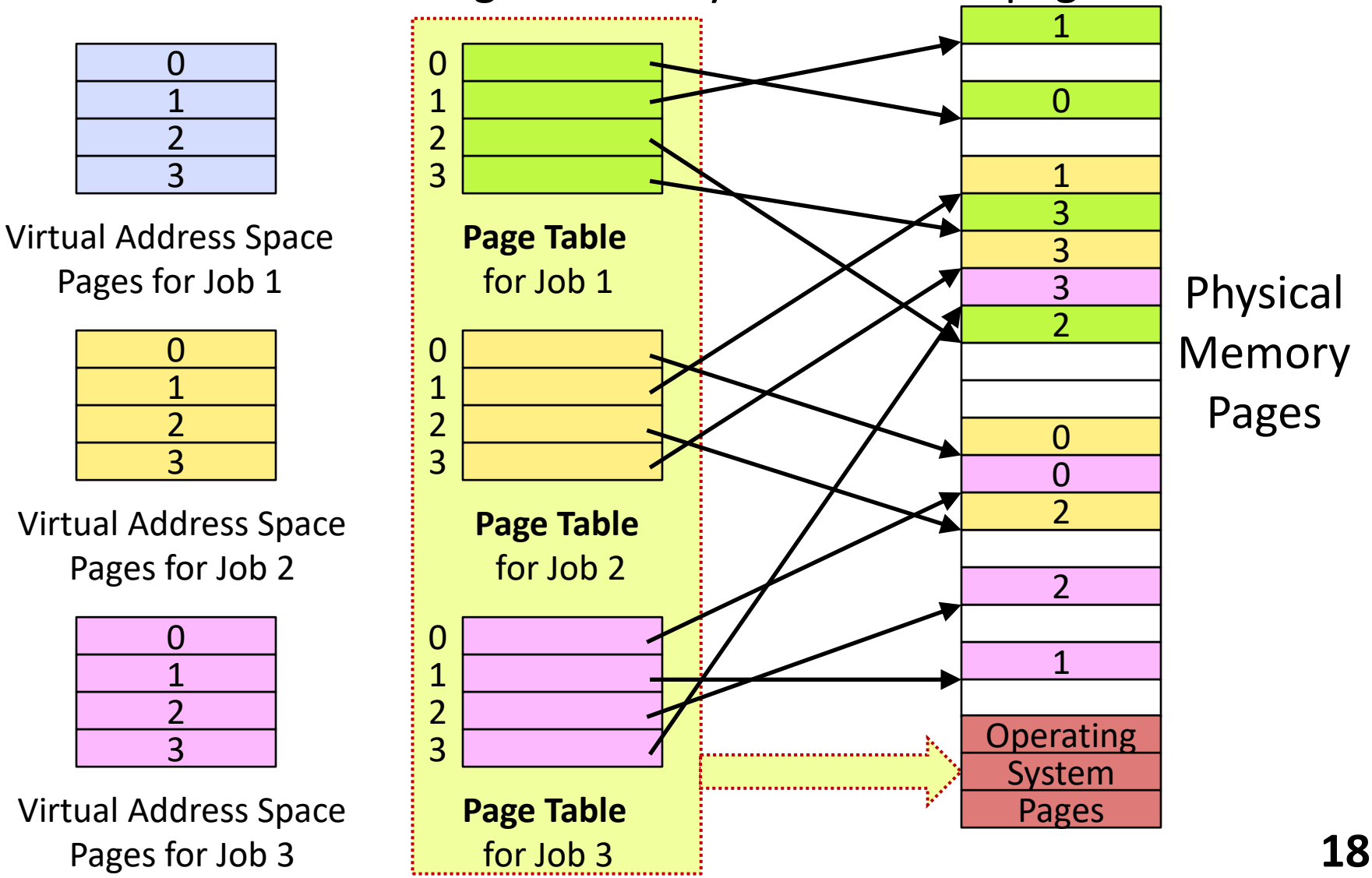
- Program-generated virtual address is split into **{VPN + offset}**



- Paging makes it possible to store a large **contiguous** virtual memory space using **non-contiguous** physical memory pages
- For each program, an **independent** Page Table is maintained

Private Address Space per User

- Each user has a **Page Table** contains an entry for each user page
- **Persistent OS** residing in memory to control all page tables



Paging Simplifies Memory Allocation

- Fixed-size pages can be kept on **OS free list** and allocated as needed to any process
- Process memory usage can easily **grow and shrink** dynamically
- Paging suffers from **internal fragmentation** *(inside Page)* where not all bytes on a page are used
  - Much less of an issue than external fragmentation or compaction for common page sizes (4-8KB)
  - But could be problematic when many CPUs change to **larger page sizes (e.g. 1MiB page)**

Coping with Limited Primary Storage

- Paging reduces fragmentation, but still face problems when a program would not fit into **primary memory** (DDR), and has to move data to/from **secondary storage** (drum, disk)

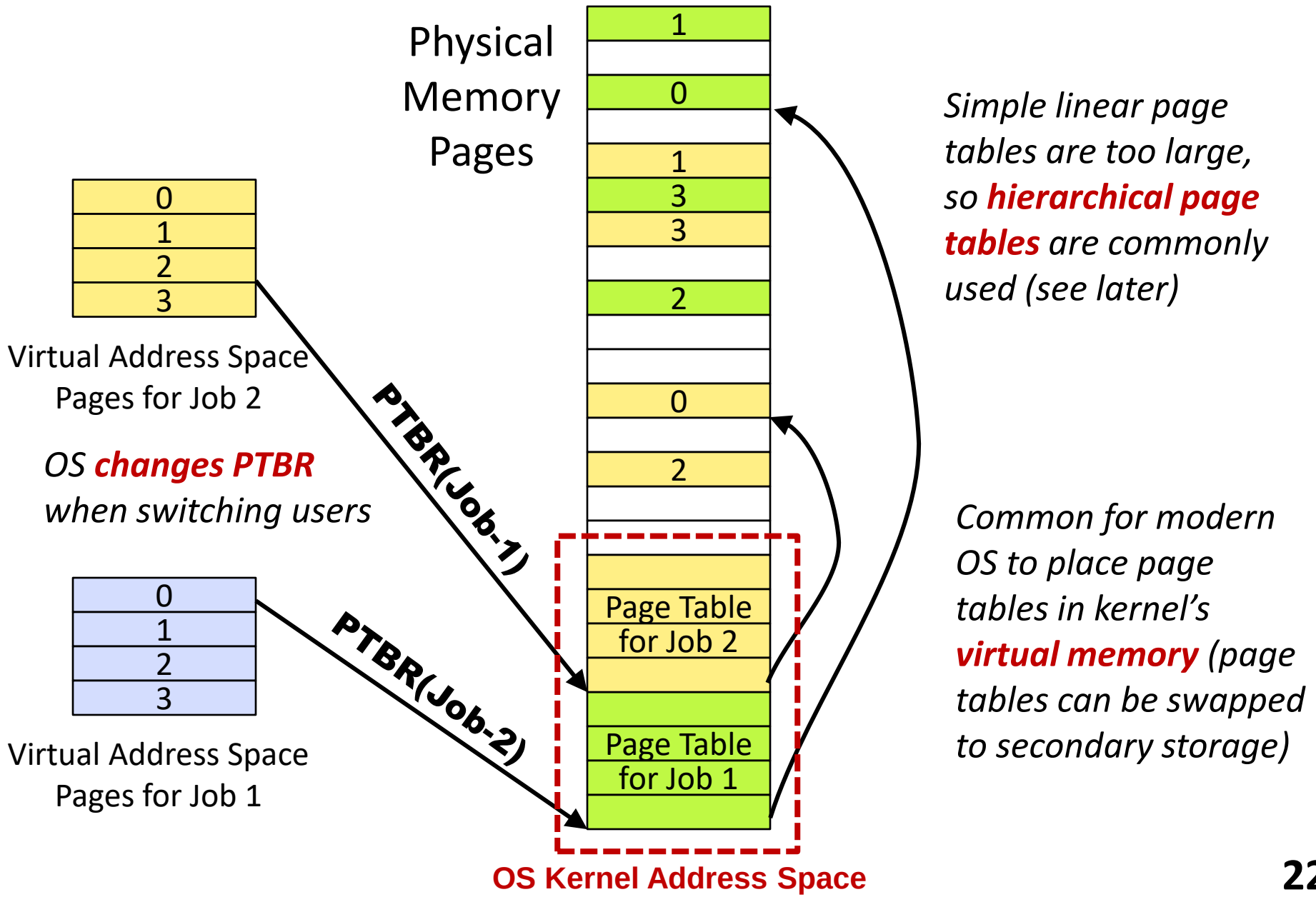
- Early approach:
  - **Manual overlays**, programmer explicitly copies code and data in and out of primary memory
    - Tedious coding, error-prone (jumping to non-resident code?)
    - IBM **Cell** microprocessor using in Playstation-3 had explicitly managed local store!
    - Many new "**deep learning**" accelerators have similar arch.

- Using virtual memory pages:
  - Put a physical page in **primary or secondary** storage wherever suitable
  - Maintain its position in a virtual memory **page table entry**.

Where should Page Tables Reside ?

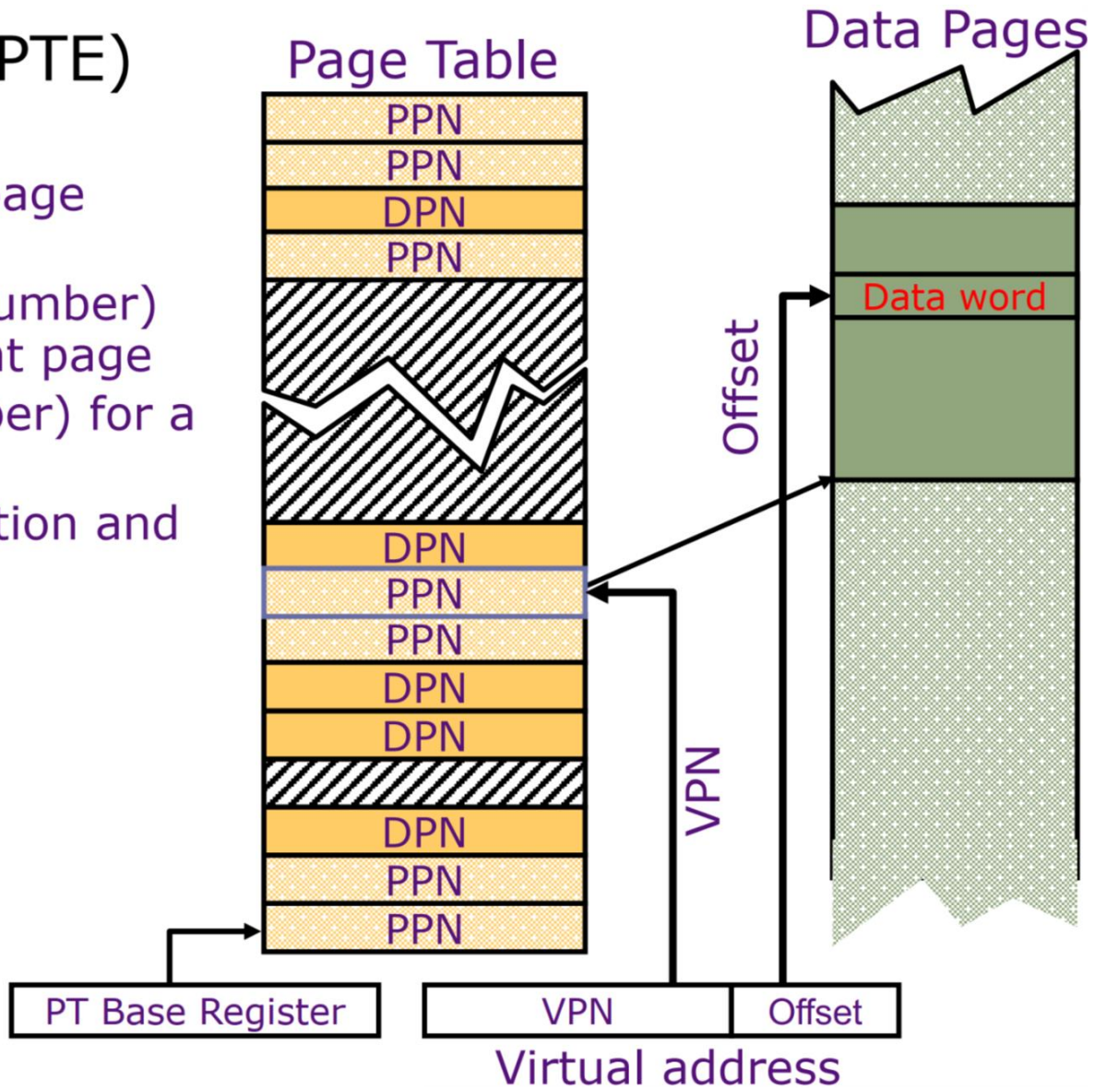
- Space required by the page table (PT) is proportional to the address space, number of users, …
  - Space requirement is large (**4GB** space = **1M** PT Entries of 4KB pages, **each user** = **4MB** for 32-bit entry)
- Bad Idea: Keep PT of current user in special registers
  - May not be feasible for large page tables
  - Too expensive to keep in registers
  - Increases the cost of context swap
- Good Idea: Keep PTs in the main memory
  - Use one **Page Table Base Register** (**PTBR**) to hold PT's location in the main memory
  - Needs **one additional reference** to read PT (to retrieve the page base address) and another to access the actual data
  - **Double the number of memory references!**

Page Tables Live in Memory

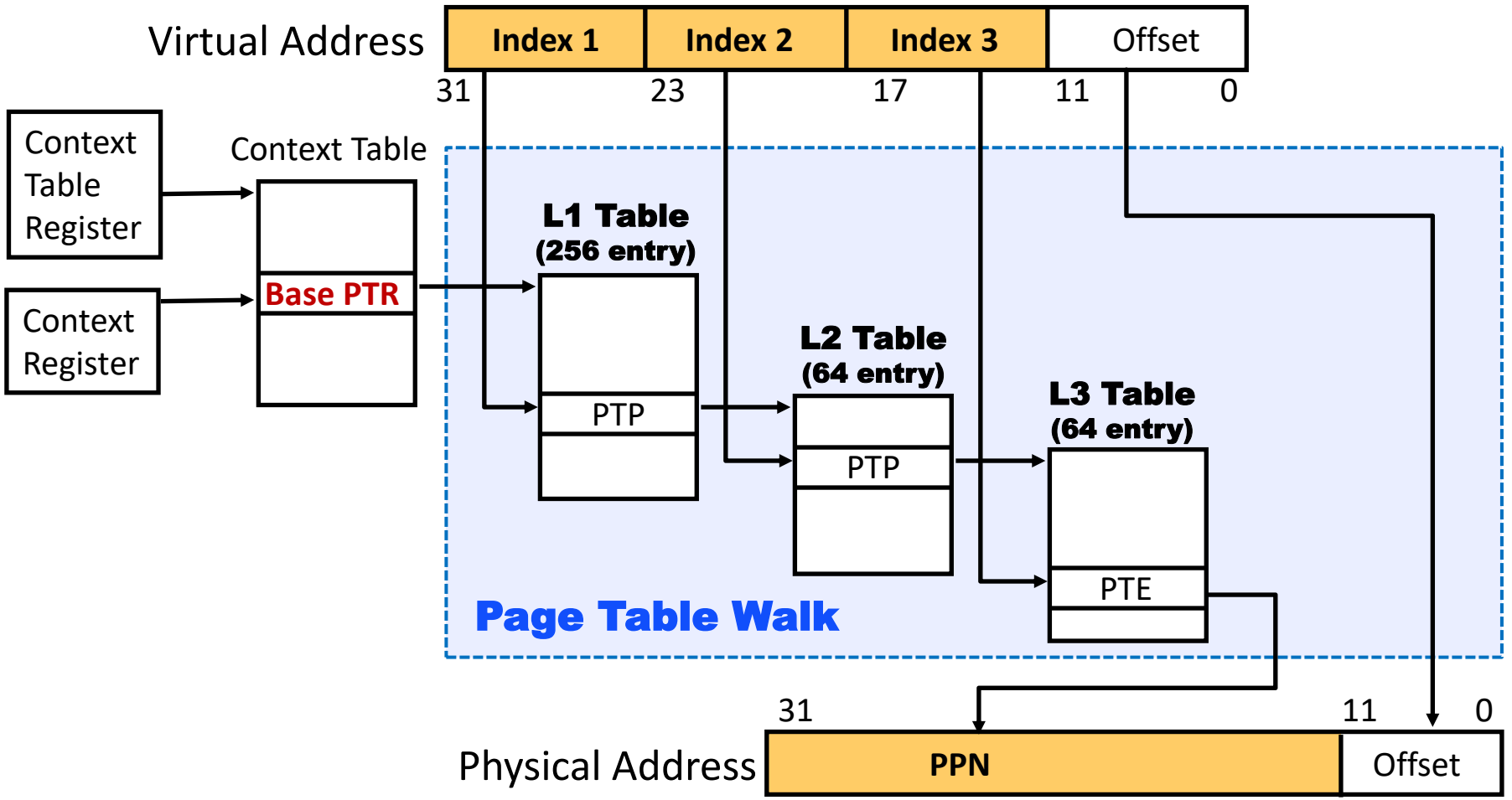


Linear Page Table

- Page Table Entry (PTE) contains:
  - A bit to indicate if a page exists
  - PPN (physical page number) for a memory-resident page
  - DPN (disk page number) for a page on the disk
  - Status bits for protection and usage



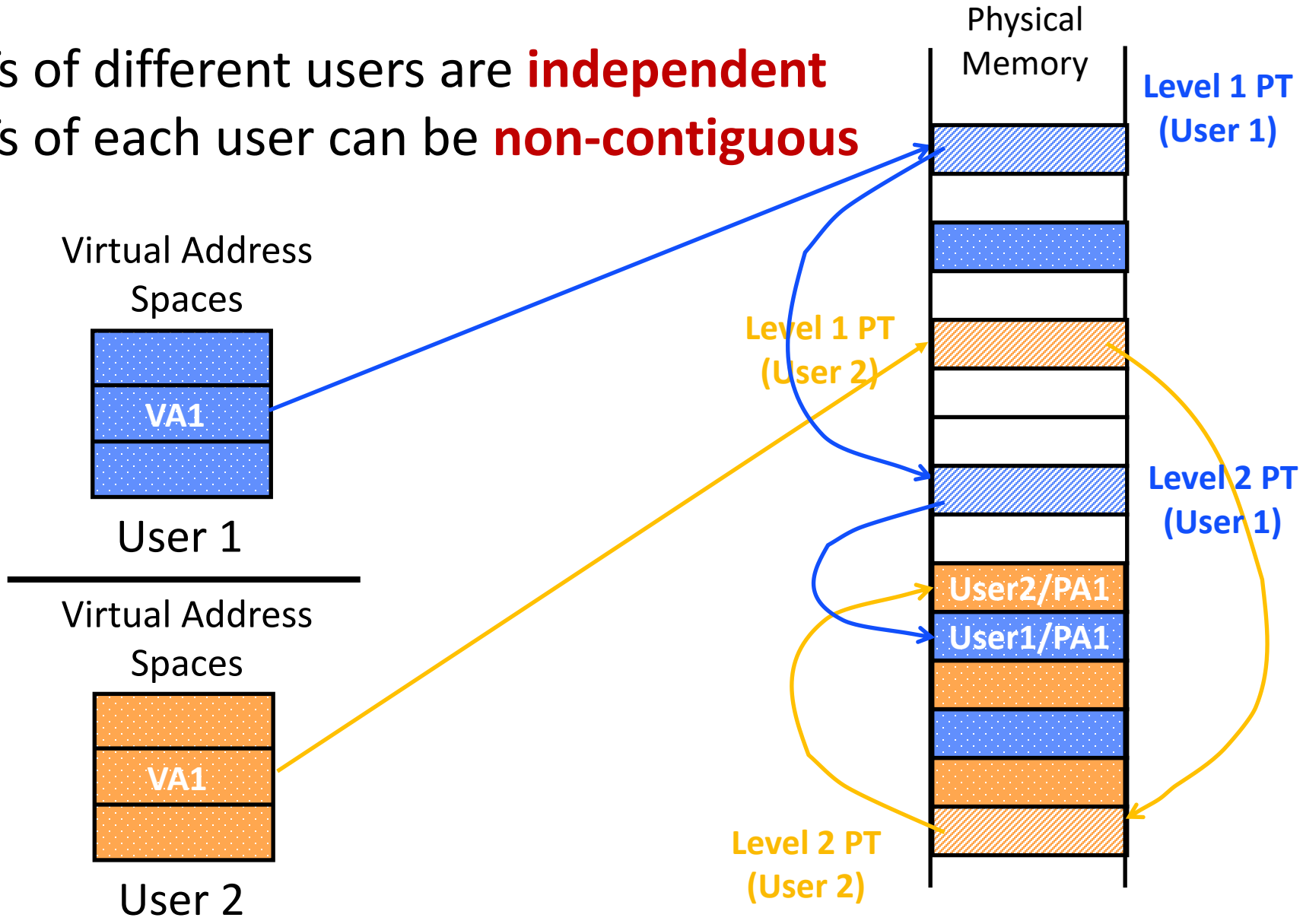
Hierarchical Page Table Walk: SPARC v8



- Hierarchical Page Table is a **tree** structure, PTBR is the **root**.
- Only create page table when necessary, **reduces memory footprint**
- Termed as **page table walk** , usually performed in **hardware unit.**

Two-Level Page Tables in Physical Memory

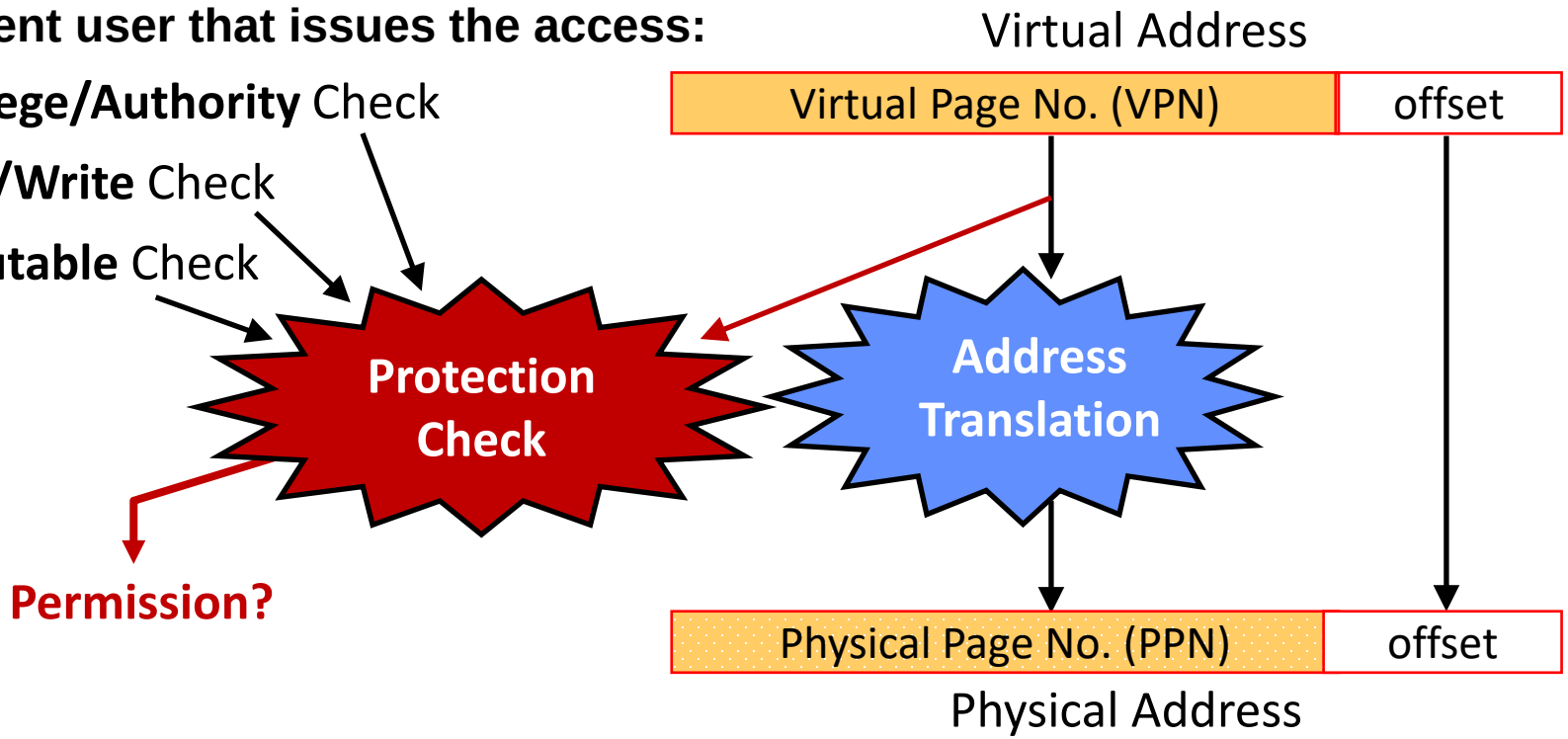
- PTs of different users are **independent**
- PTs of each user can be **non-contiguous**



Address Translation & Protection Check

For current user that issues the access:

- **Privilege/Authority** Check
- **Read/Write** Check
- **Executable** Check

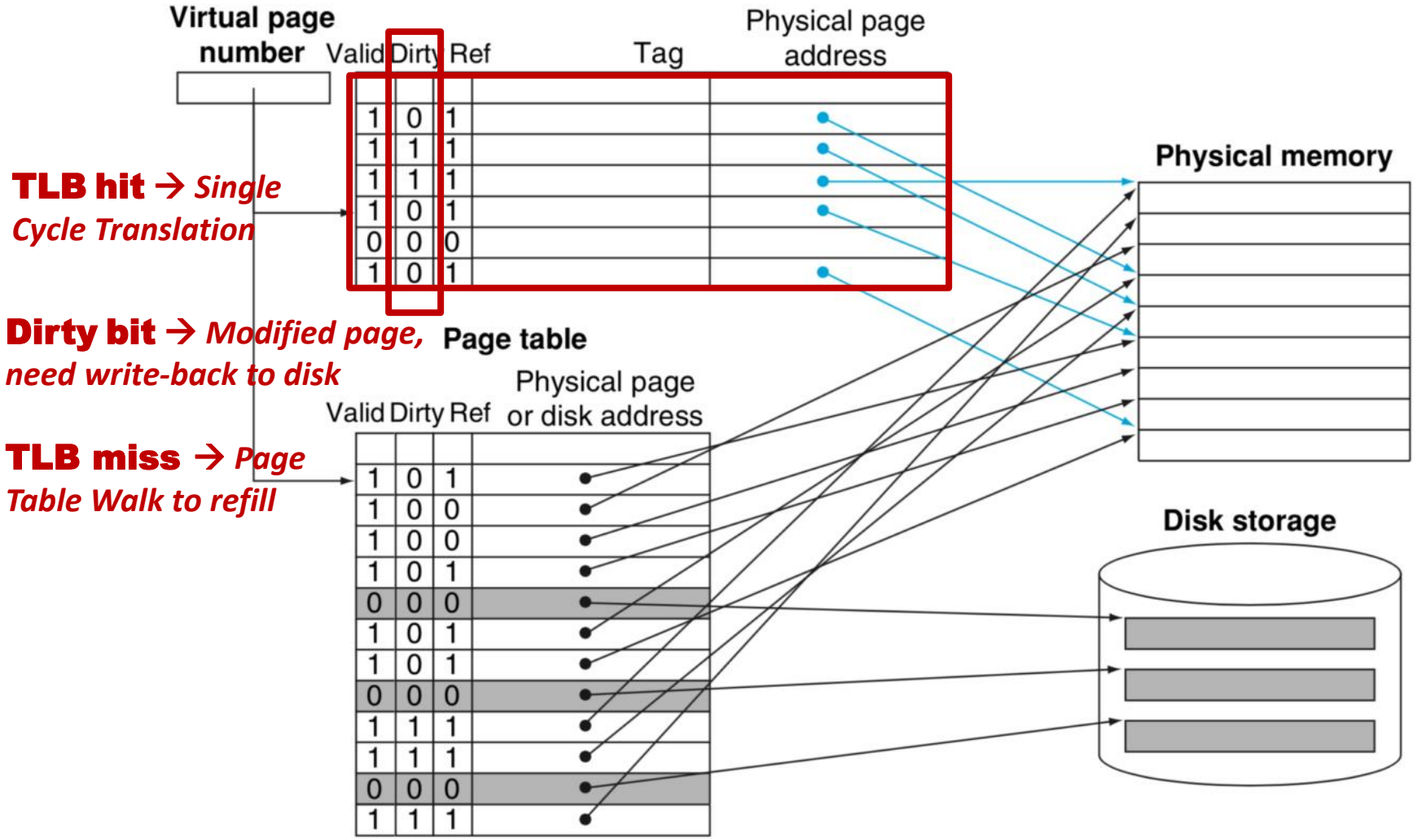


- Every instruction/data access needs address translation and **protection checks**
- Address translation is very **expensive**!
  In a **multiple-level** page table, each reference requires several memory accesses

Translation-Lookaside Buffers (TLB)

*A good VM design needs to be **fast** (~ one cycle) and space **efficient***

Idea: ***Cache** the address translation of **frequently used** pages*

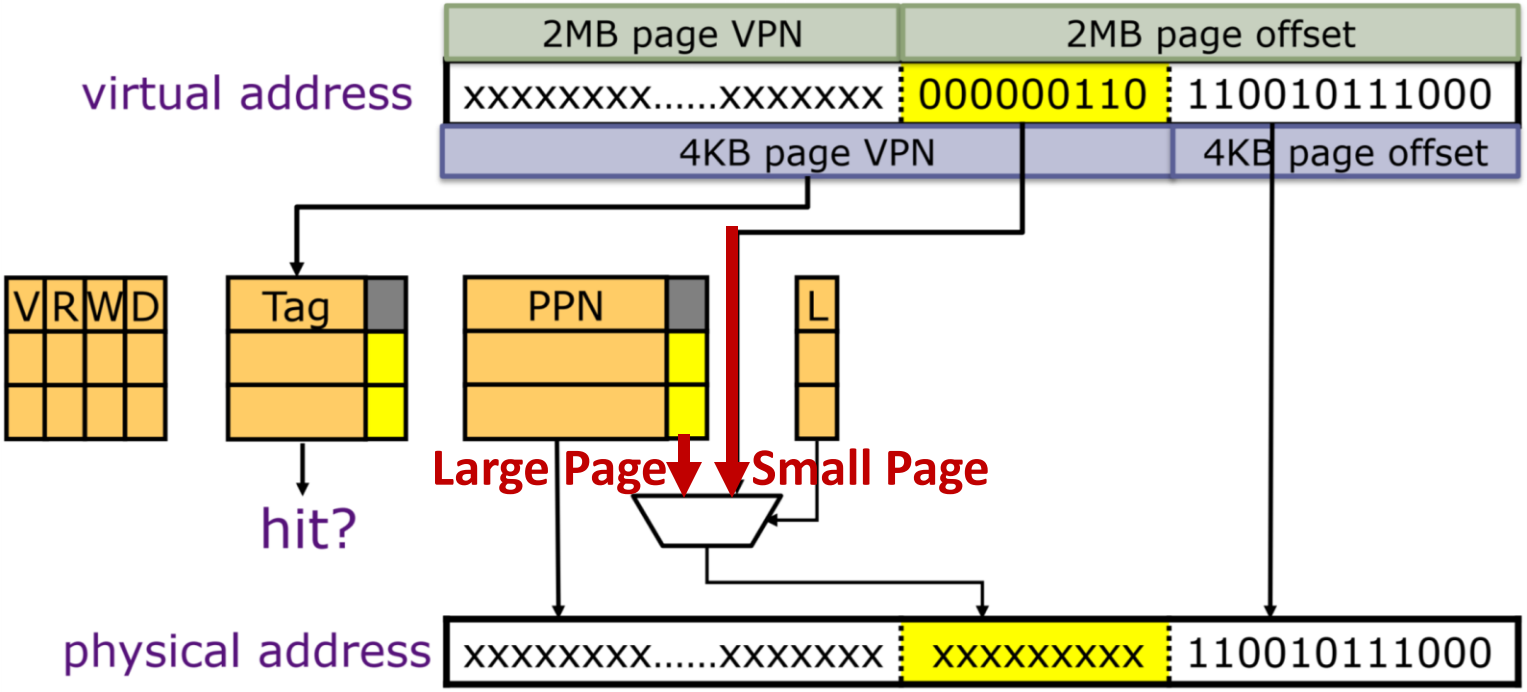


TLB Designs

- Typically **32-128** entries, usually **fully associative**
  - Each entry maps a large page, hence **less spatial locality** across pages ➔ more likely that two entries conflict
  - Sometimes **larger TLBs** (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have **multi-level TLBs** (L1 and L2)
- **Replacement policy:** Random, FIFO or LRU
- **TLB Reach**: Size of largest virtual address space that can be simultaneously mapped by TLB
  - Example: 64 TLB entries, 4KB pages, one page per entry
  - TLB Reach = ***64 entries * 4 KB = 256 KB (if contiguous)***?

Variable-Size Page TLB

Some Systems support multiple page sizes (managed by OS)



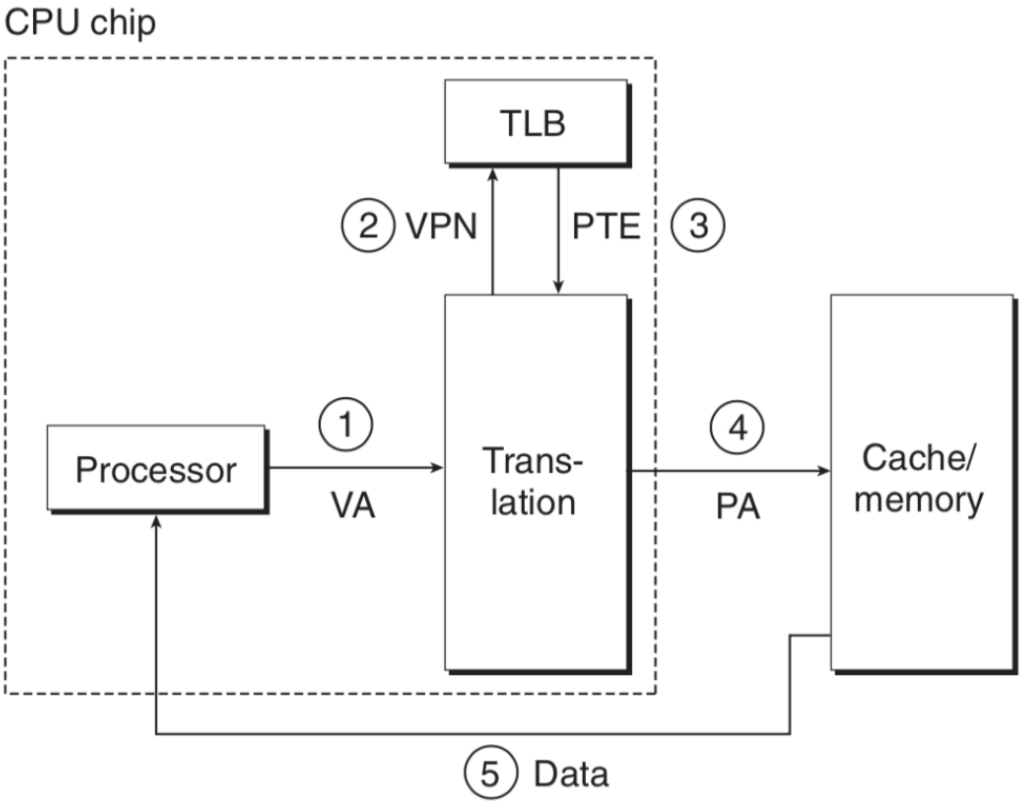
Page mask	Page Size	Pagemask	Page Size
0_0000_0000	4KB	0_0011_1111	256KB
0_0000_0011	16KB	0_1111_1111	1MB
0_0000_1111	64KB	1_1111_1111	2MB

*MIPS using **Pagemask** mark different Page size (OS manage)*

Handling a TLB Miss

- TLB misses when:
  - The page exists in memory → just add the missing entry in TLB.
  - The page doesn't exist in memory→ transfer control to the **OS**
- Software (*MIPS, Alpha*)
  - TLB miss causes an **TLB miss exception** and the OS walks the page tables and reloads TLB.
  - **Very expensive** on out-of-order superscalar processor as requires a flush of pipeline to jump to trap handler.
- Hardware (*SPARC v8, x86, ARM, PowerPC, RISC-V*)
  - A **memory management unit (MMU)** walks the page tables and reloads the TLB.
  - If a **missing page** (data or PT) is encountered during the TLB reloading, MMU gives up and signals a **Page Fault exception** for the original instruction.

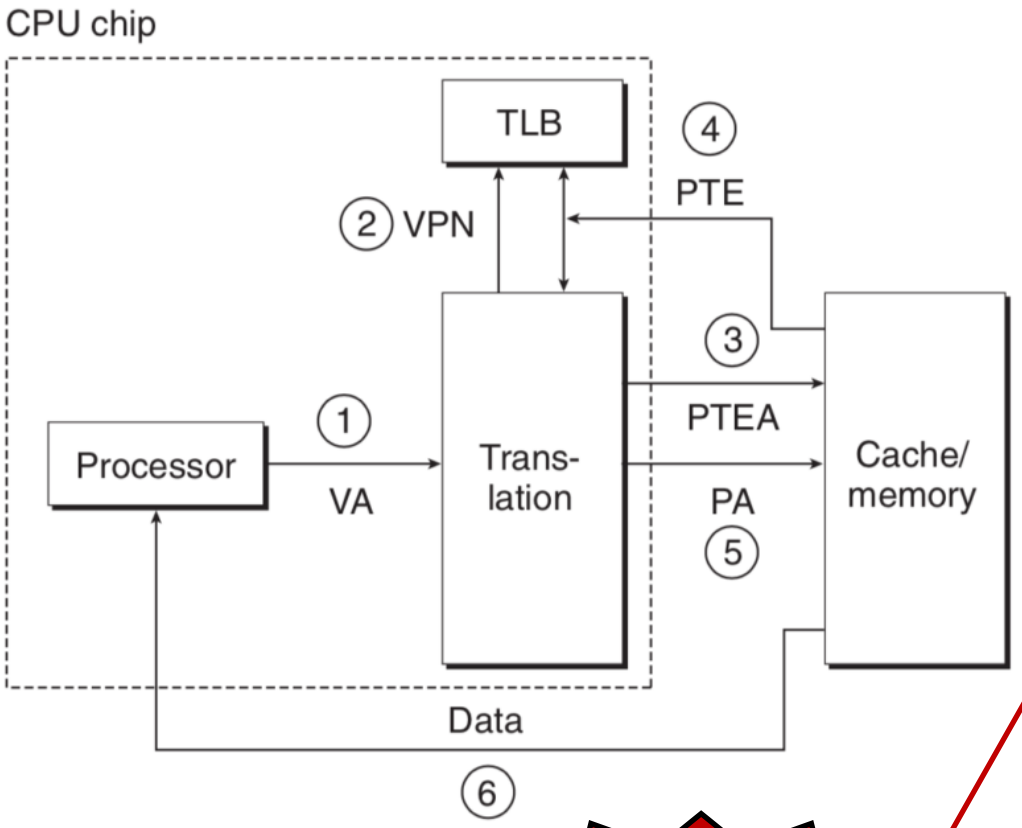
Handling a TLB Miss



TLB Hit

① Processor sends Virtual Address (VA)
② Extract Virtual Page Number(VPN) from VA. Query TLB using VPN.
③ TLB returns Page Table Entry (PTE).
④ Combine PTE with Page Offset to get Physical Address (PA). Query Cache using PA.
⑤ Send data to processor

Handling a TLB Miss



What if there is no valid PT Entry?

TLB Miss

① Processor sends Virtual Address (VA)

② Extract Virtual Page Number (VPN) from VA. Query TLB using VPN.

③ TLB miss. Query Memory using Page Table Entry Address (PTEA) to get Page Table Entry (PTE)

④ Save VPN to PTE mapping in TLB.

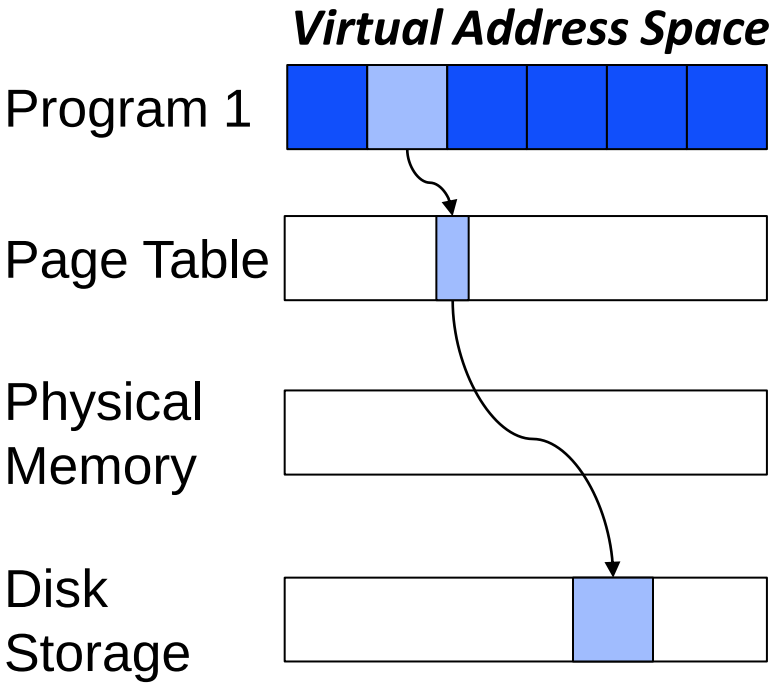
⑤ Combine PTE with Page Offset to get Physical Address (PA). Query Cache using PA.

⑥ Send data to processor.

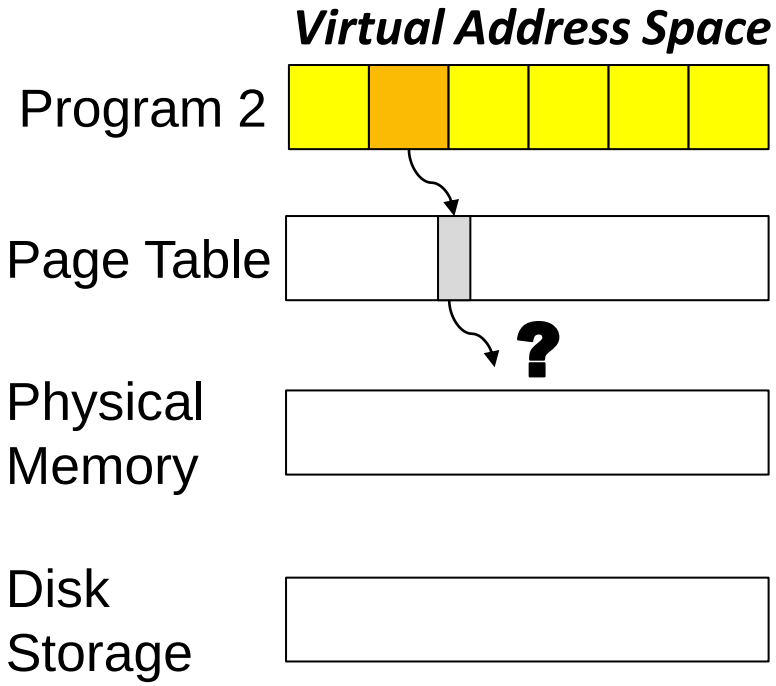
Page Fault Exception

- Occurs when an instruction references a memory page that is **not in main memory.**

Case 1: Page is swapped to secondary storage (e.g. disk)



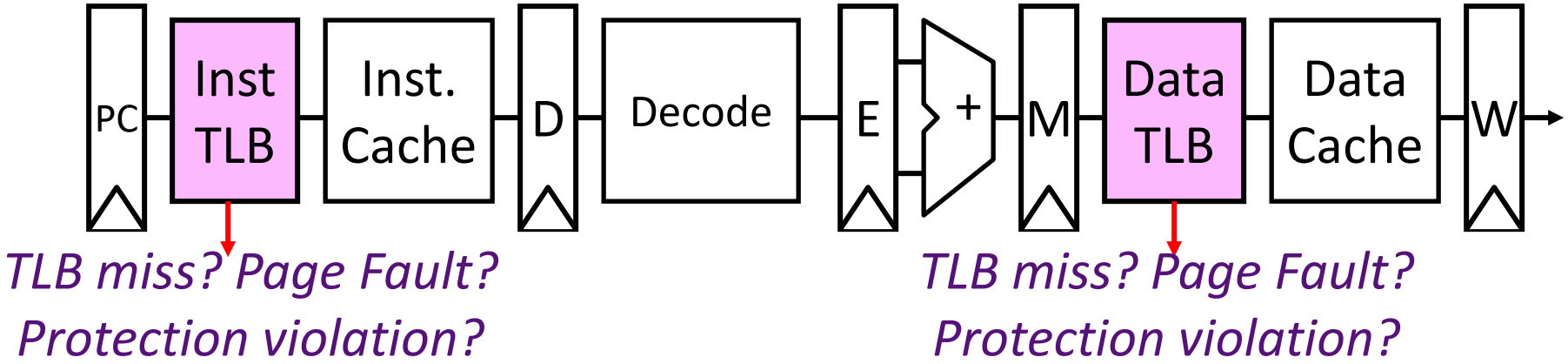
Case 2: Page is virtually allocated but not really created (e.g. malloc)



Page Fault Exception

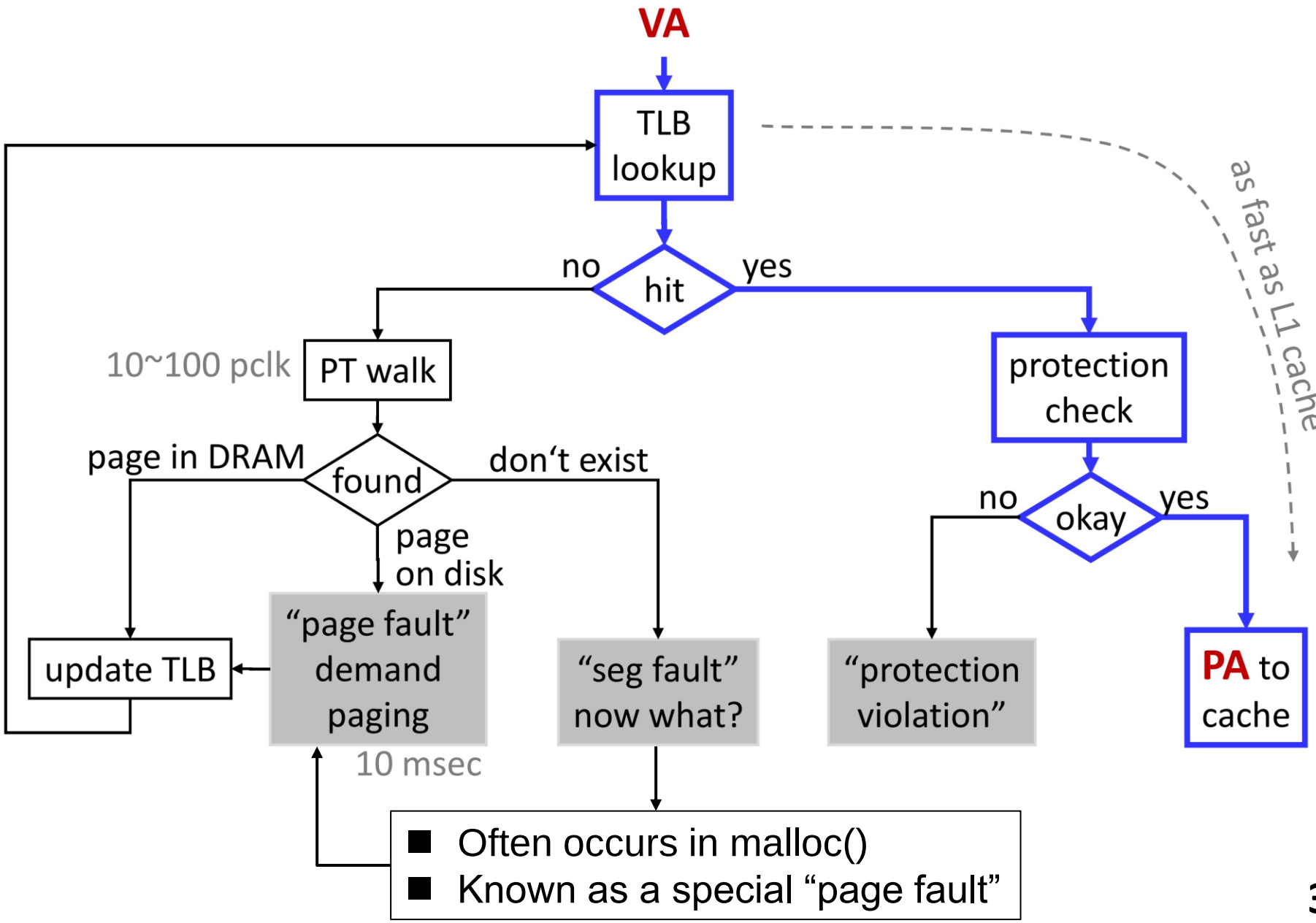
- Since it takes a **long time** to transfer a page (in msecs), page faults are handled completely in software **by OS**
- **Page Fault Handler (of OS)** does the following:
  - – Assign an unused page in DRAM
    - **If no unused page is left**, a page currently in DRAM is **swapped out**
      - **Replace** using Pseudo-LRU, implemented in software
      - **Write-back** to disk if the replaced page is 'dirty'
      - **Update PTE** of that VPN->PPN as invalid/DPN
  - – If virtual page exist in disk, Initiate transfer of the requested page from disk to DRAM, storing in the newly assigned page
    - Another job may be run on the CPU while the first job waits for the requested page to be read from disk
  - – Page table entry of the requested page is updated with a (now) valid PPN
  - – Return and re-execute the exception-causing instruction
    - Need for precise exceptions

Handling VM-related exceptions

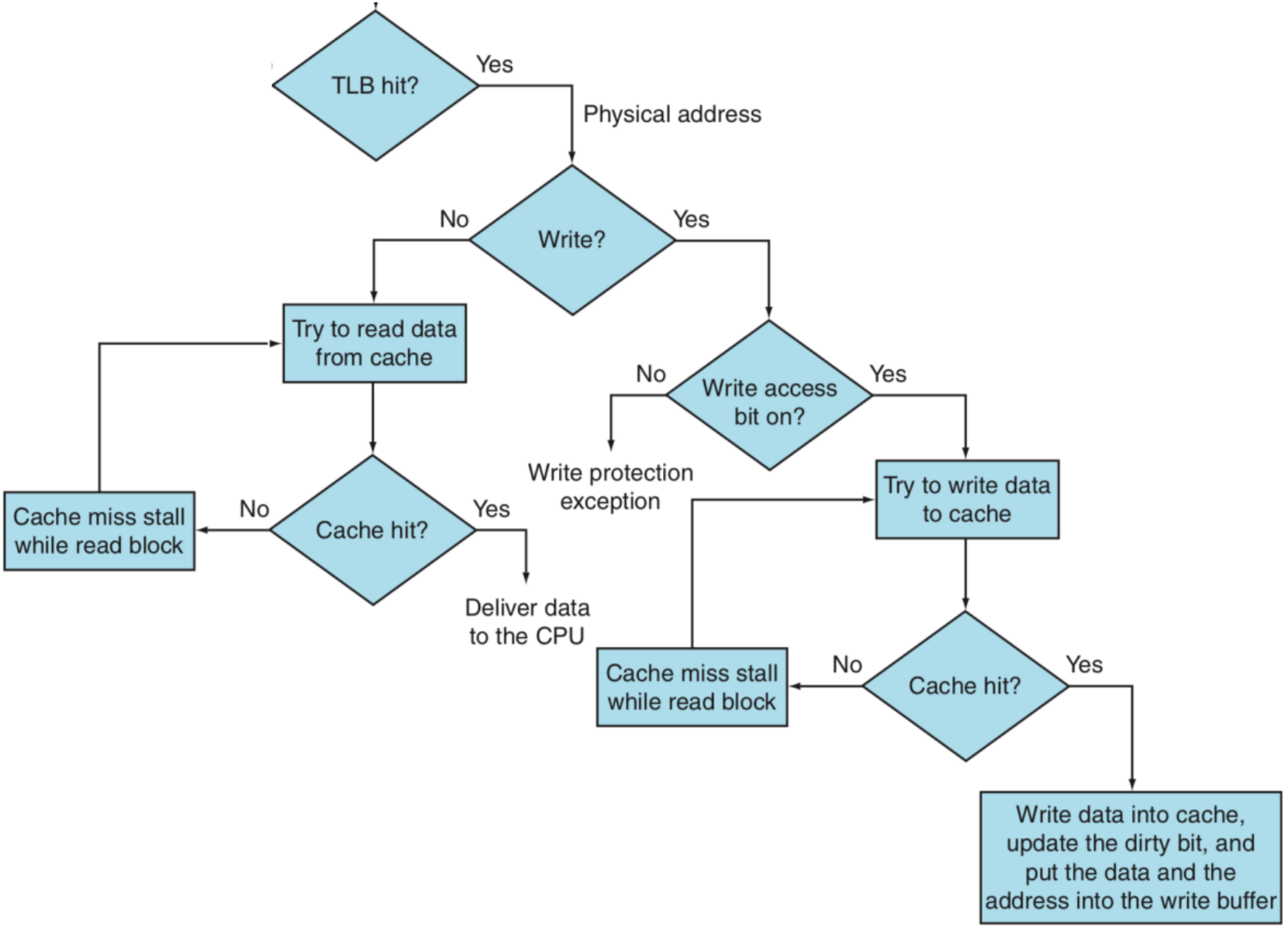


- Handling **TLB miss** needs a hardware or software mechanism to refill TLB（usually fulfilled by hardware as MMU）
- Handling **Page Fault** (e.g., page is on disk) needs *restartable* exception so software handler can resume after retrieving page
  - **Precise exceptions** are easy to **restart**
  - Can be imprecise but restartable, but this complicates OS software
- A **protection violation** may abort process
  - But often handled the same as a page fault

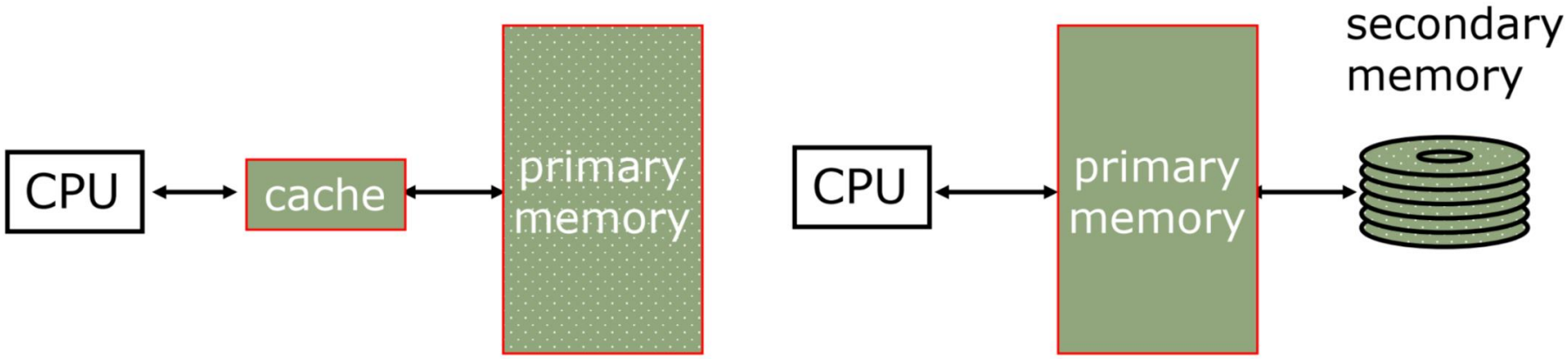
Address Translation – Putting it all together



Address Translation – Putting it all together



Summary: Caching v.s Demand Paging



Caching	*Demand paging*
cache entry	page frame
cache block (~32 bytes)	page (~4K bytes)
cache miss rate (1% to 20%)	page miss rate (<0.001%)
cache hit (~1 cycle)	page hit (~100 cycles)
cache miss (~100 cycles)	page miss (~5M cycles)
a miss is handled in *hardware*	a miss is handled mostly in *software*

Summary: TLB, Cache and Page

TLB、 Page Table、 D-Cache and Physical Page (Total 2^4=16 cases)

- IF TLB hit **or** D$ hit → PTE valid
- IF PTE valid → Physical Page exists in DDR

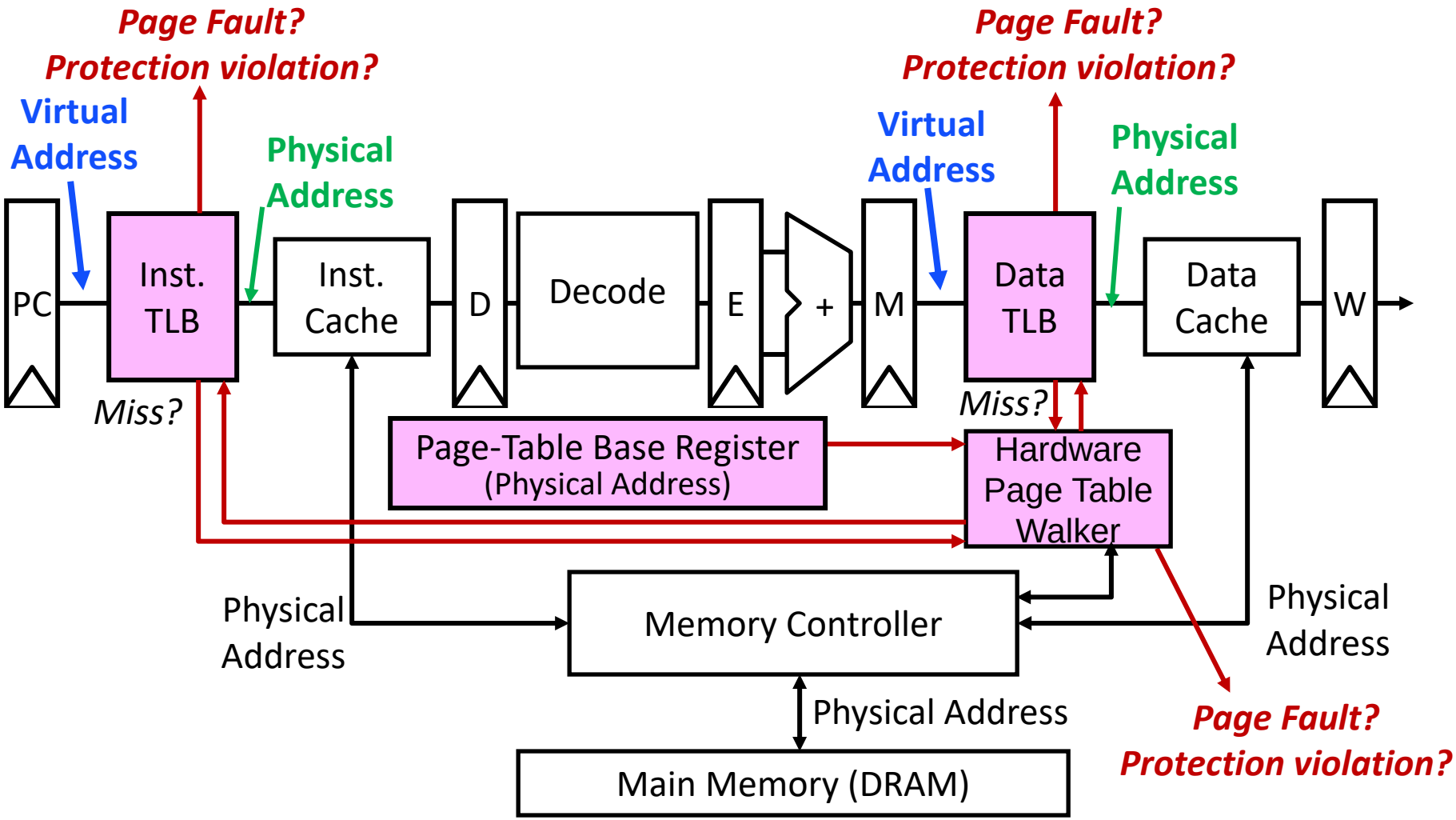
Cost ↓

Situation	TLB	Page Table Entry	D-Cache	Physical Page	Operations
D$ hit	**Hit**	**If TLB hits, PTE valid**	**Hit**	**If D$ hits, page exists**	No need to check Phys. Mem
D$ miss	**Hit**	**If TLB hit, PTE valid**	**Miss**	**If TLB hits, page exists**	Update D$
TLB miss	**Miss**	**If D$ hit, PTE valid**	**Hit**	**If D$ hits, page exists**	Update TLB, access TLB again, then access D$
TLB+D$ both miss	**Miss**	**Hit**	**Miss**	**If PT hits, page exists**	Update TLB/D$, then access TLB/D$ again
Page Fault	**Miss**	**Miss**	**Miss**	**Miss**	Page Fault process

Summary: TLB, Cache and Page

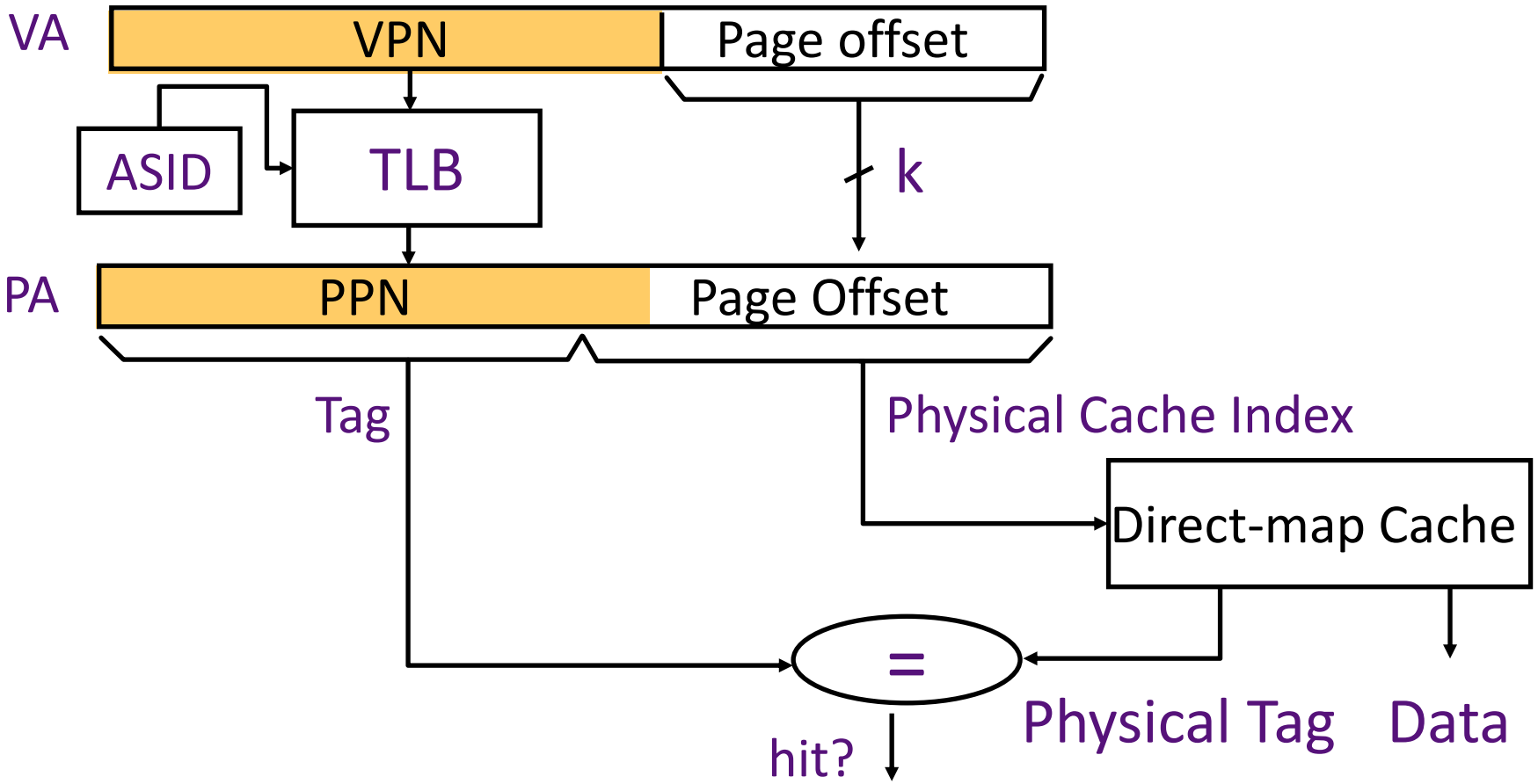
TLB	PAGE TABLE	CACHE	POSSIBLE?	NOTES
Hit	Hit	Miss	Yes	Page Table is never checked if TLB hits
Miss	Hit	Hit	Yes	TLB misses, entry found in Page Table. After retry data is found in Cache
Miss	Hit	Miss	Yes	TLB misses, entry found in Page Table. After retry data misses in Cache
Miss	Miss	Miss	Yes	TLB misses, followed by a Page Fault. After retry data misses in Cache
Hit	Miss	Miss	No	Cannot have translation in TLB if Page Table not in memory
Hit	Miss	Hit	No	Cannot have translation in TLB if Page Table not in memory
Miss	Miss	Hit	No	Data cannot be allowed in Cache if Page Table is not memory

Page-Based Virtual-Memory Machine
(Hardware Page-Table Walk)



- **Caches** are accessed using **translated physical addresses**
- **MMU directly uses physical address** to access page tables

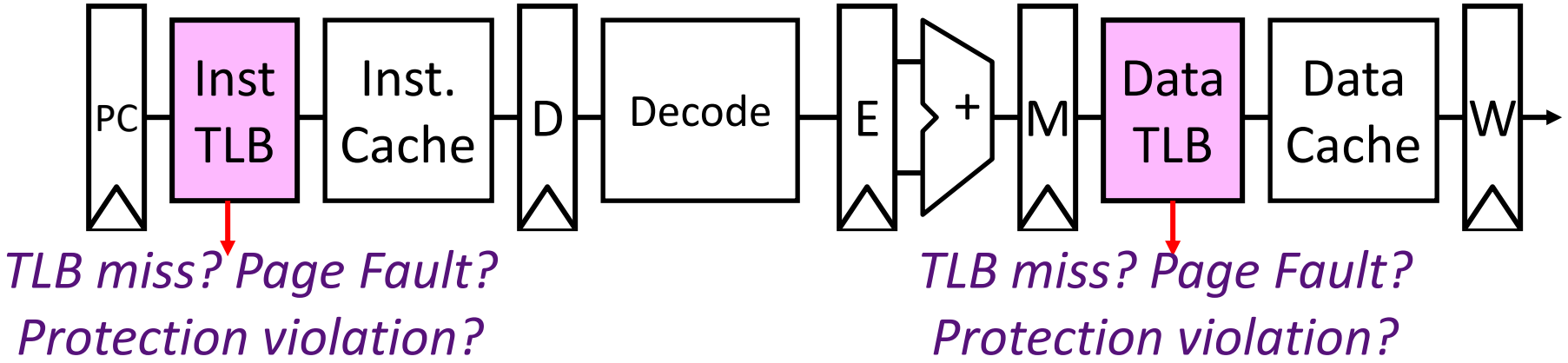
Sequential Access to TLB & Cache (Physical Index/Physical Tag, PIPT)



Adding **one more stage** for TLB access will increase:

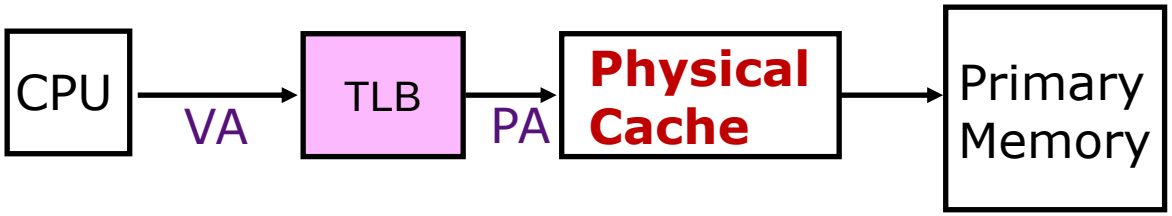
- For I-Cache: more branch misprediction penalty
- For D-Cache: more load latency (**critical path!**)

Address Translation in CPU Pipeline

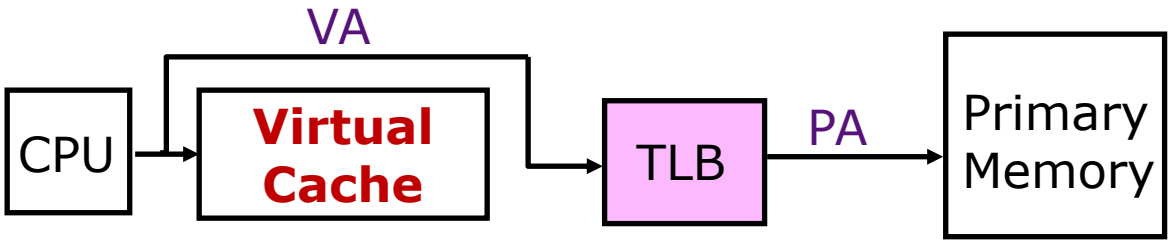


- Need to cope with additional latency of TLB:
  - Slow down the clock?
    - Unacceptable for modern CPUs
  - Pipeline the TLB and cache access?
    - Sub-optimal solution (still long latency)
  - Virtual address caches
  - Parallel TLB/cache access

Virtual-Address Caches



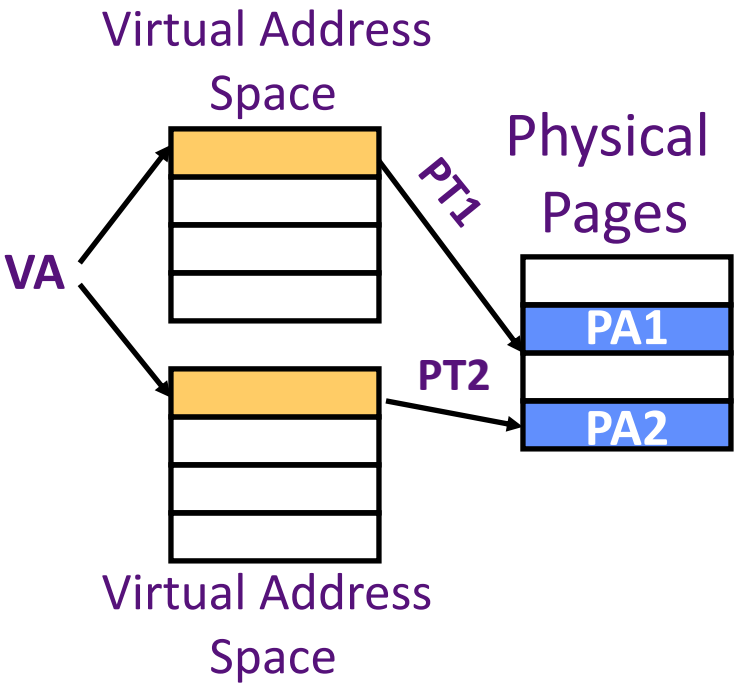
Alternative: place the cache before the TLB



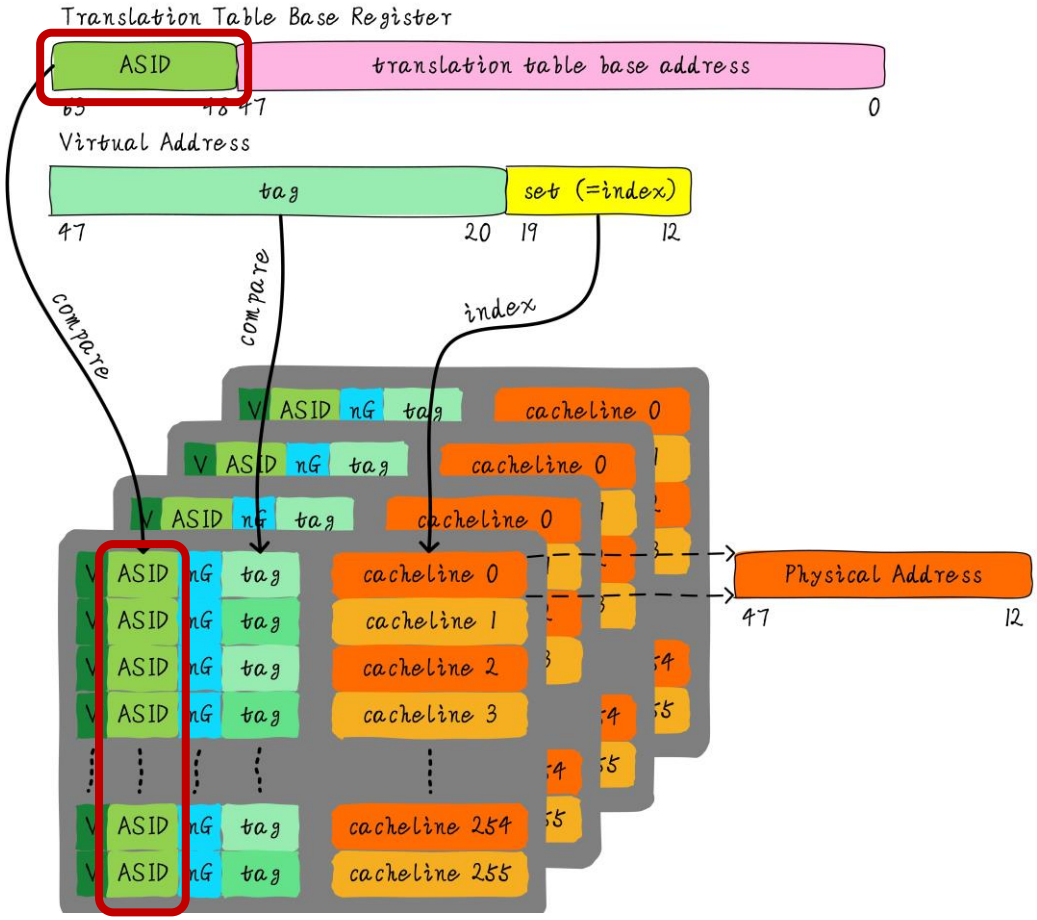
(StrongARM)

- **One-cycle latency** cache access in case of cache hit (+)
- **Homonym** and **Aliasing** problems (-)

Homonym in Virtual Address

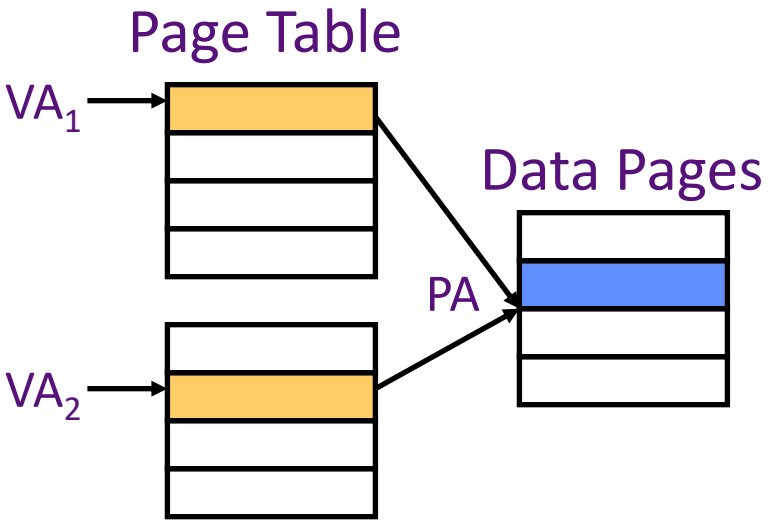


Same virtual address mapped to two physical pages

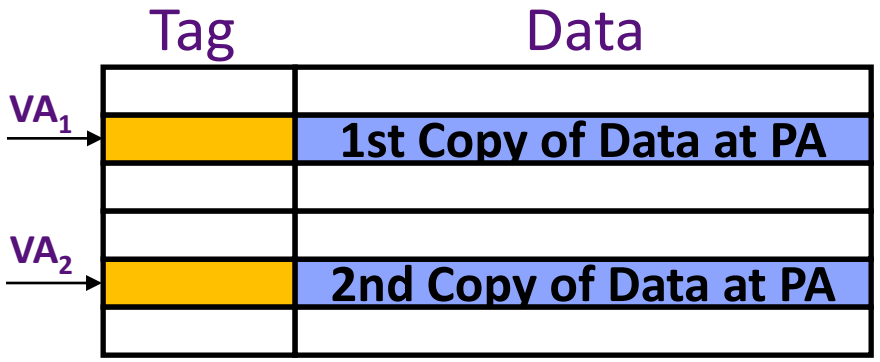


- Conflicting virtually-tagged entry (both **TLB** and VIVT **cache**)
- Software (OS): Cache and TLB needs to be flushed on **context switch**
- Hardware: add Address Space Identifier (**ASID**) into Tags

Aliasing in Virtual-Address Caches



Two virtual pages share one physical page



Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

General Solution: *Prevent aliases coexisting in cache*

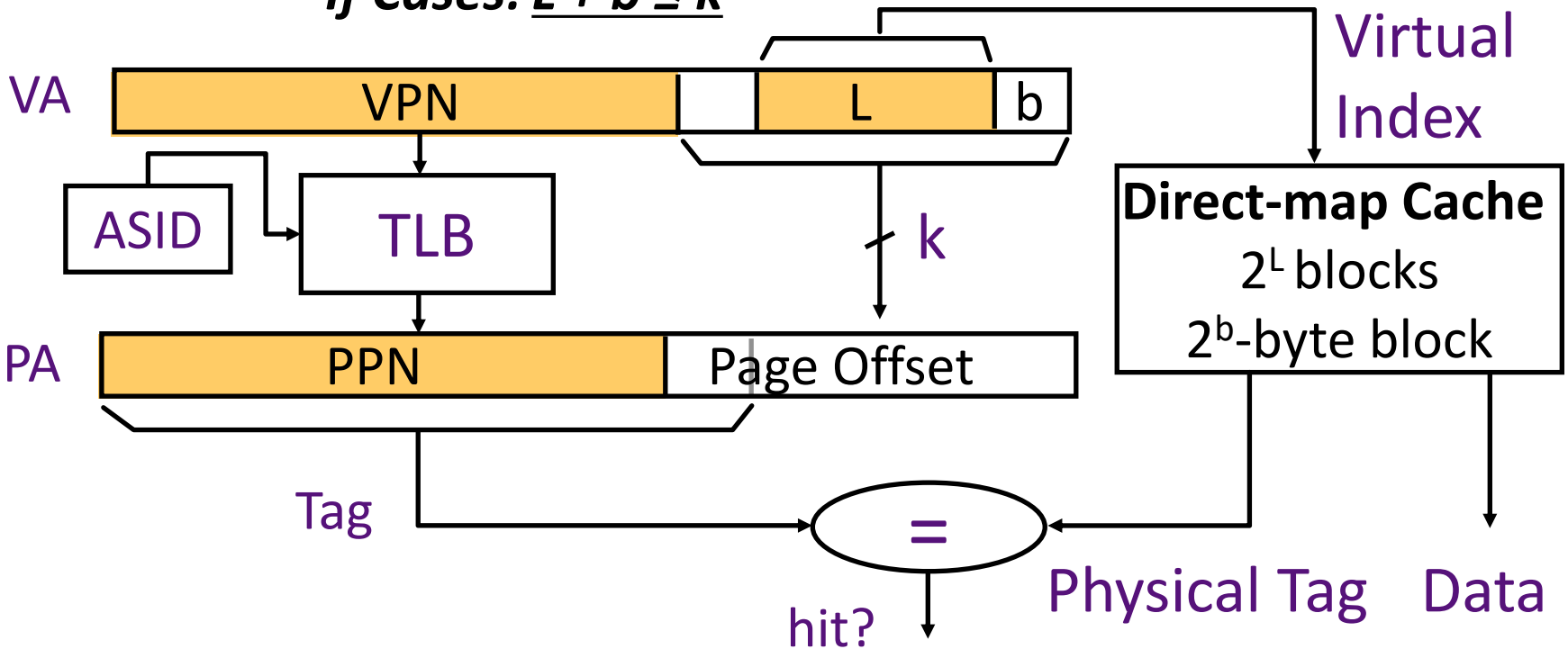
Software Solution (i.e., OS) for direct-mapped cache (early SPARCs):

VAs of shared pages must **agree in cache index bits**; this **ensures (forces) conflict** for all VAs accessing the same PA

Concurrent Access to TLB & Cache
(Virtual Index/Physical Tag, VIPT)

L=Cache Index, b=Cache Block, k=Page Size,

If Cases: $L + b \leq k$



Index L-bits of VA is available without consulting the TLB.

→ Cache and TLB accesses can begin **simultaneously**!

→ Tag comparison is made after both accesses are completed.

→ Actually, these are still **Physical Index Cache**

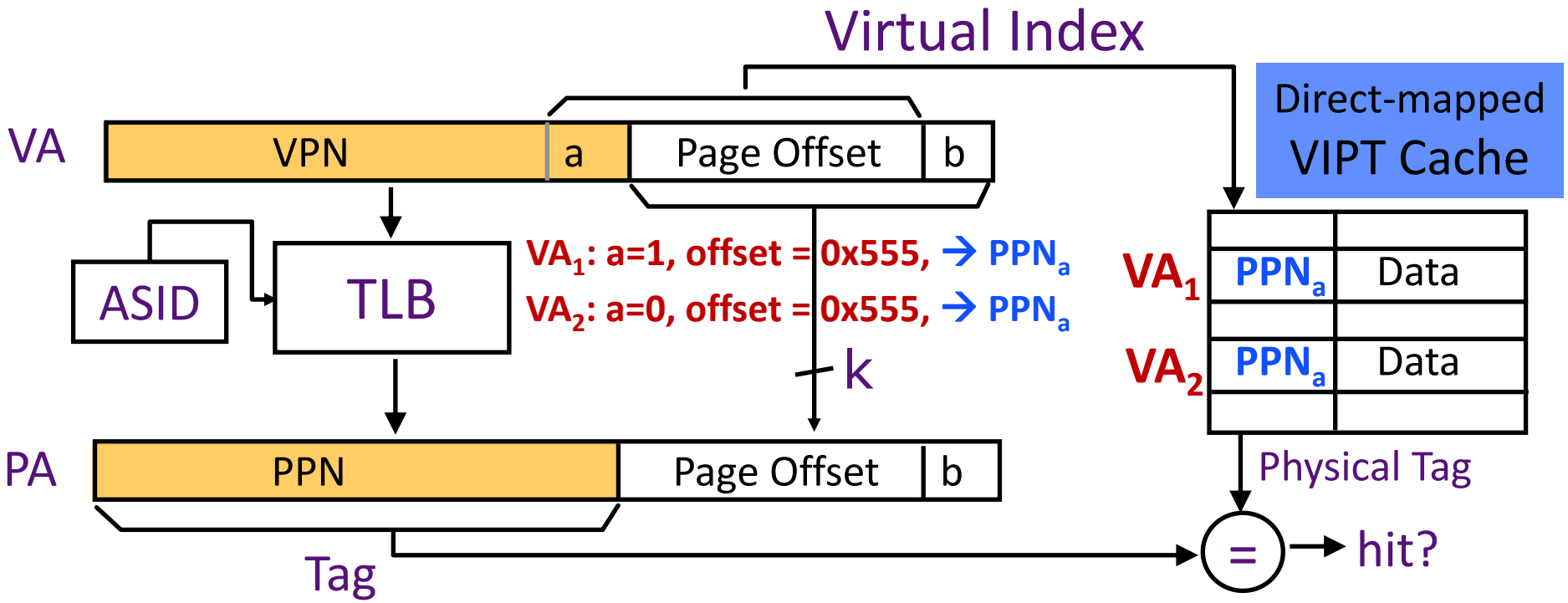
Concurrent Access to TLB & Large L1

The problem with L1 Cache > Page size

L=Cache Index, b=Cache Block, k=Page Size,

If Cases: $\underline{L + b > k}$

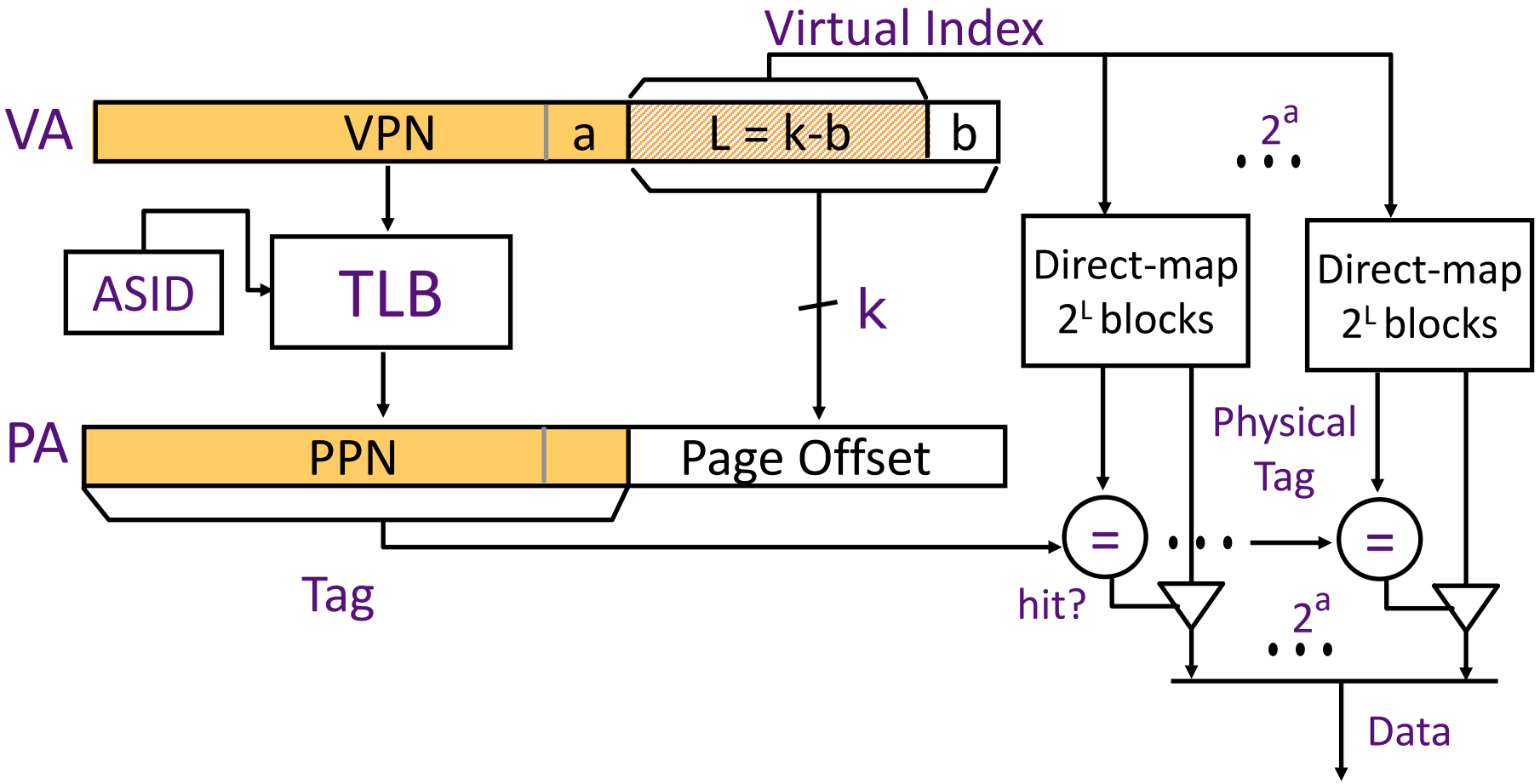
Aliasing: ($VA_1 \neq VA_2$) *both map to* ***the same*** **PA**



What if $VA_1.a \neq VA_2.a$?

Virtual-Index Physical-Tag Caches:

Associative Organization

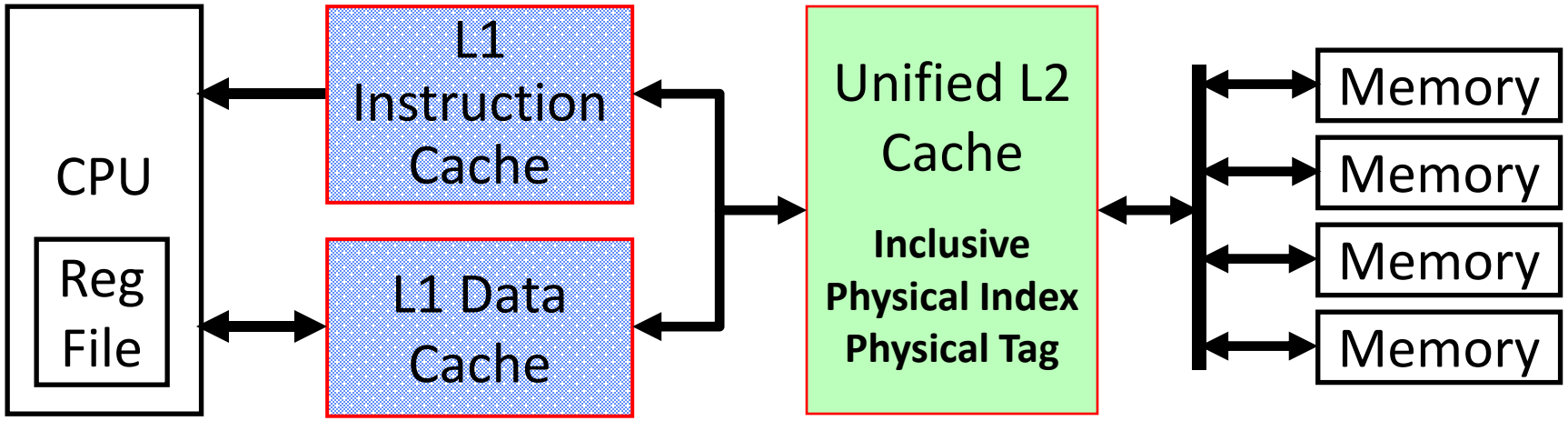


After the PPN is known, 2^a physical tags are compared

→ e.g. 32 KB L1 Cache = 8-way associative (Intel)

How does this scheme scale to larger caches? Not Good

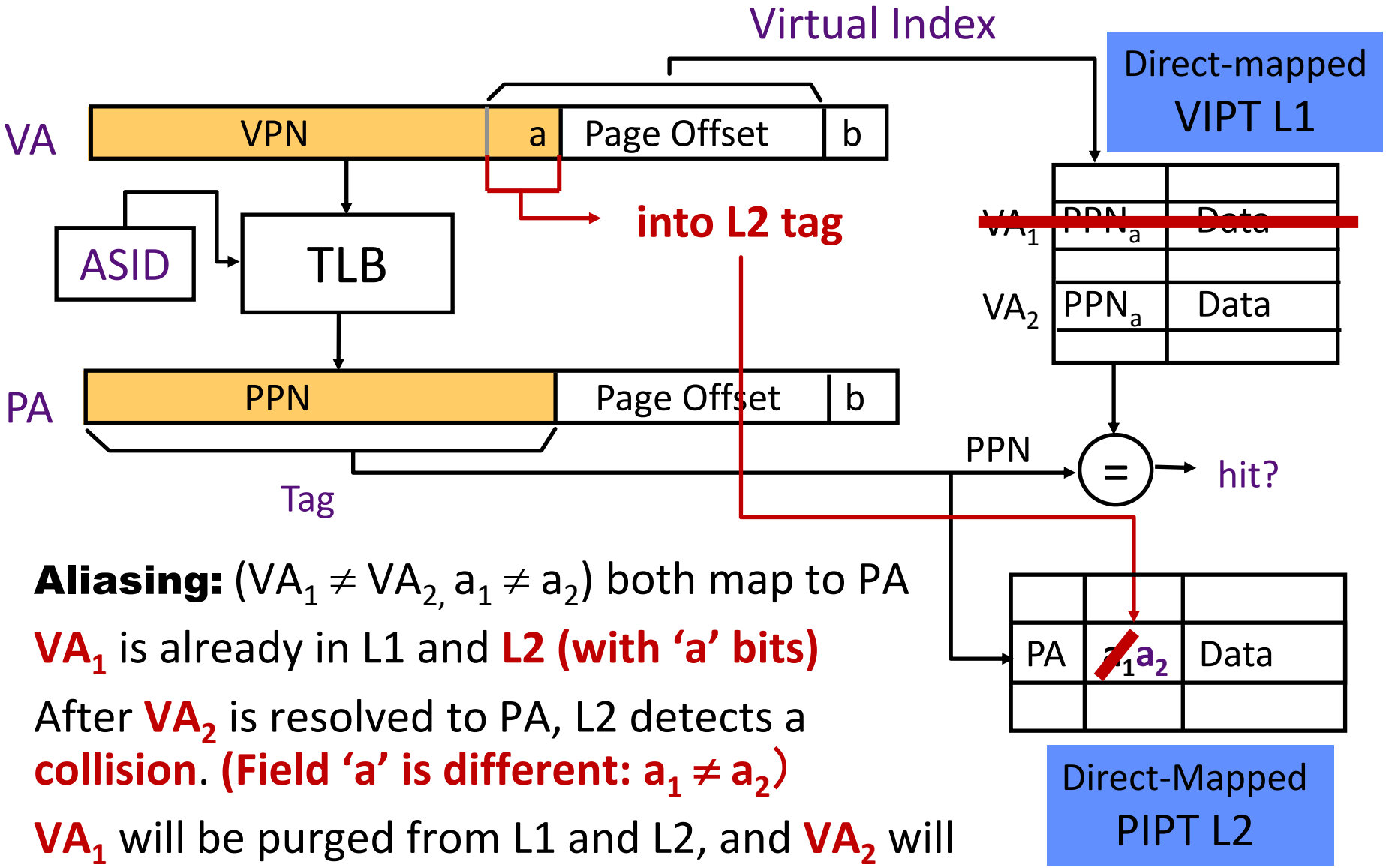
A solution via Second-Level Cache



Physical-index physical-tag (PIPT) , Inclusive L2 Cache:

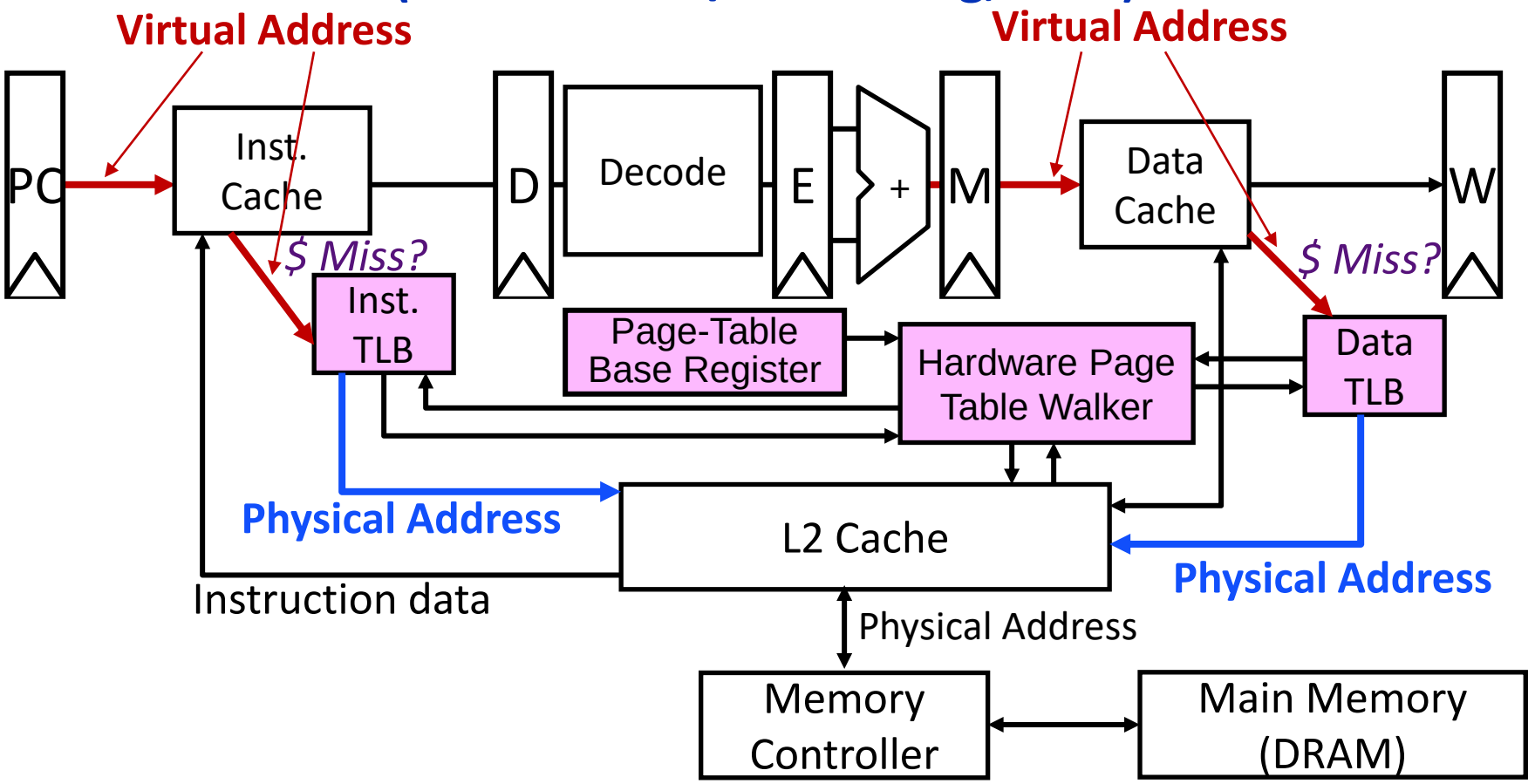
- **PIPT**: Applied for most L2/L3/LLC cache design
- **Unified**: L2 cache backs up both Instruction and Data L1 caches
- **Inclusive:** L2 has copies of all cache lines in both L1 D-Cache and I-Cache

Anti-Aliasing (VIPT) Using L2 [*MIPS R10000,1996*]



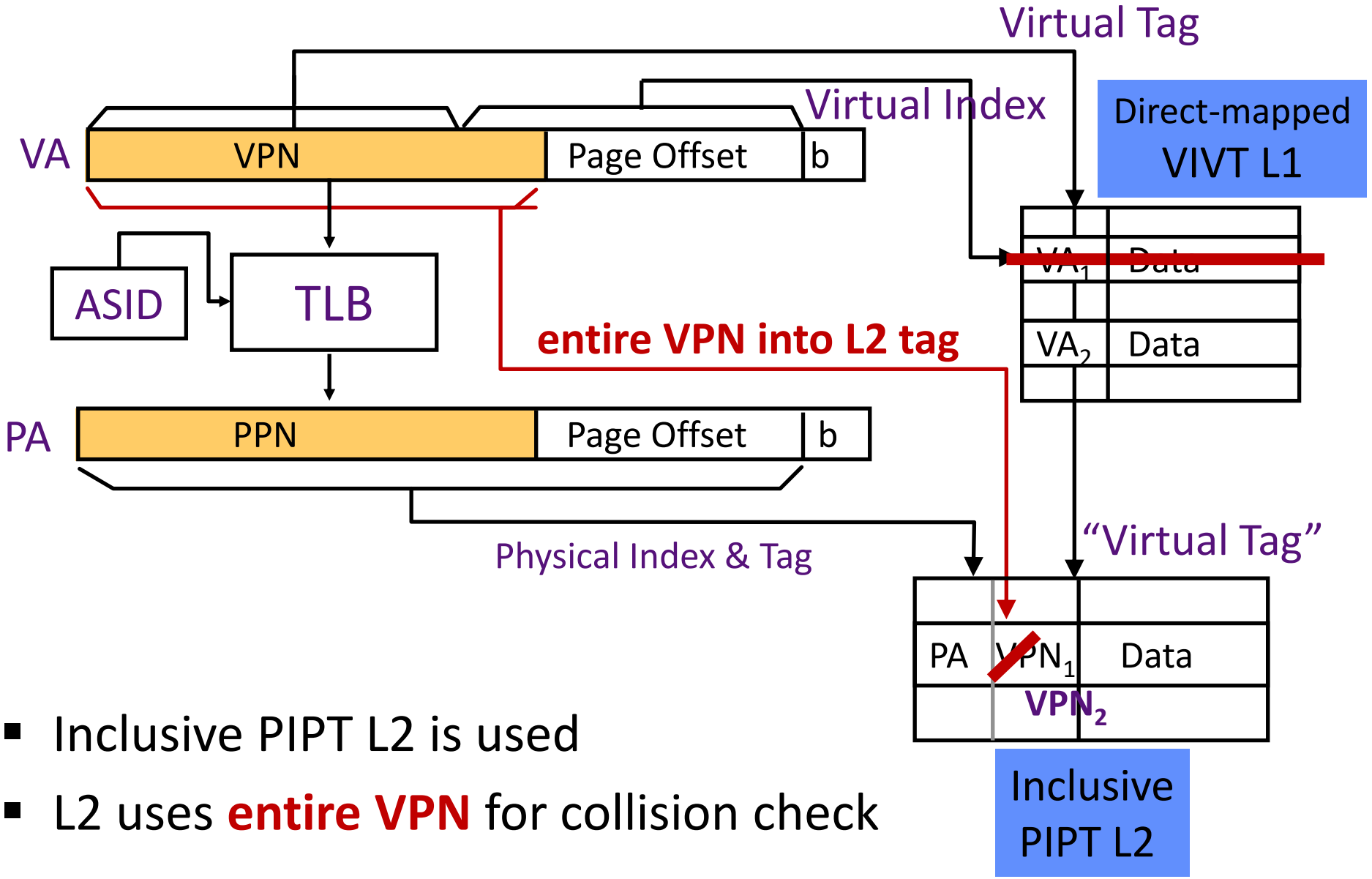
- **Aliasing:** ($VA_1 \neq VA_2$, $a_1 \neq a_2$) both map to PA
- **VA_1** is already in L1 and **L2 (with 'a' bits)**
- After **VA_2** is resolved to PA, L2 detects a **collision**. **(Field 'a' is different: $a_1 \neq a_2$)**
- **VA_1** will be purged from L1 and L2, and **VA_2** will be loaded $\Rightarrow$ *no aliasing !*

Virtually Addressed Cache
(Virtual Index/Virtual Tag, VIVT)



- Directly use **full virtual address** for L1 cache access
- Only check TLB on **L1 cache miss**
- Use **physical address** for L2 cache access

Anti-Aliasing (VIVT) using L2



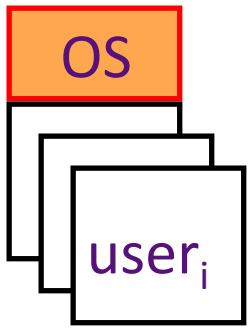
- Inclusive PIPT L2 is used
- L2 uses **entire VPN** for collision check

Summary：Modern Virtual Memory Systems

Illusion of a large, private, uniform store

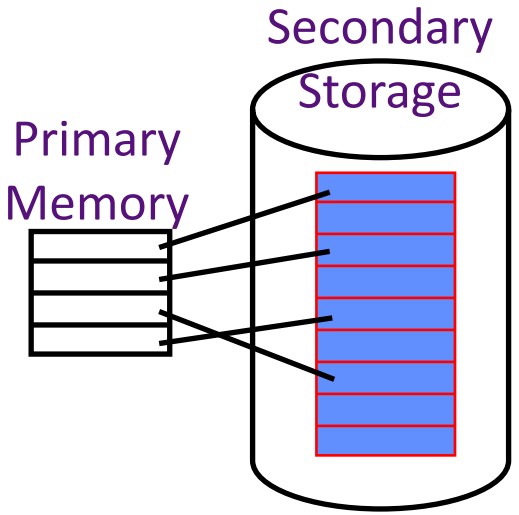
Protection & Privacy

- Several users, each with their private address space and one or more shared address spaces



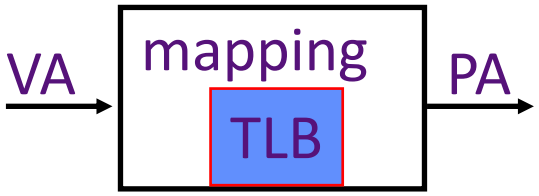
Demand Paging

- Provides the ability to run programs larger than the primary memory
- Hides differences in physical memory layouts



Cost

- The price is address translation on each memory reference
- Use **TLB** and **Virtual Indexed cache**



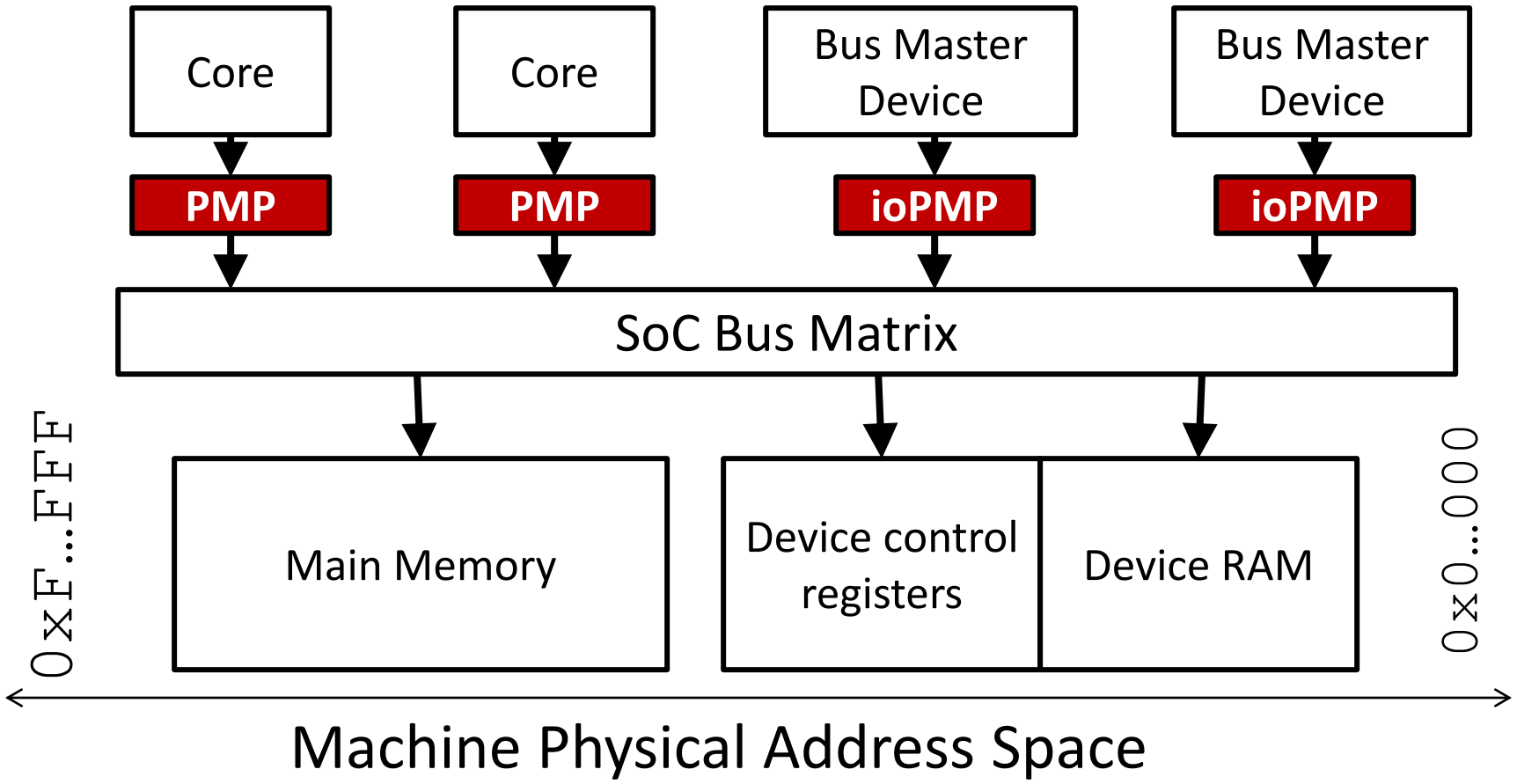
Why a Privileged Architecture?

- Profiles (Simple Embedded w/wo Protection, Unix-like OS, Cloud OS)
- Privileges and Modes
- Privileged Features
  - CSRs
  - Instructions
- Memory Addressing
  - Translation
  - Protection
- Trap Handling
  - Exceptions
  - Interrupts
- Counters
  - Time
  - Performance

RISC-V Privilege Modes

- Machine mode (M-mode, highest privileges)
  - A.K.A monitor mode, microcode mode, …
- Hypervisor-Extended Supervisor Mode (HS-Mode)
- Supervisor Mode (S-mode)
- User Mode (U-mode, lowest privileges)
- Supported combinations of modes:
  - M (simple embedded systems)
  - M, U (embedded systems with security)
  - M, S, U (systems running Unix-like OS)
  - M, S, HS, U (systems running hypervisors, Cloud OS Capable)

Physical Memory Protection (PMP)

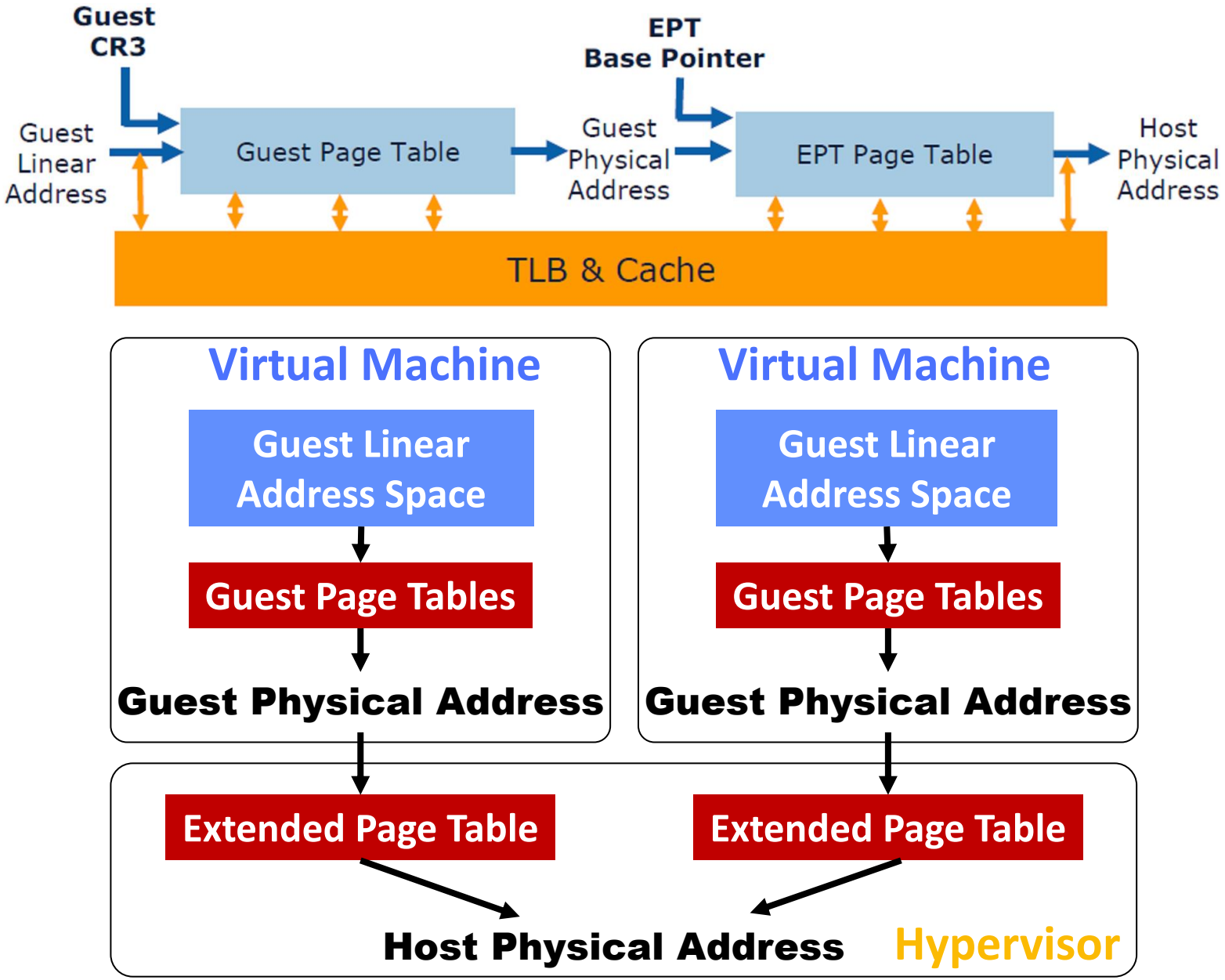


An optional **physical memory protection (PMP)** unit provides per-hart machine-mode control registers to allow physical memory access privileges (read, write, execute) to be specified for each physical memory region

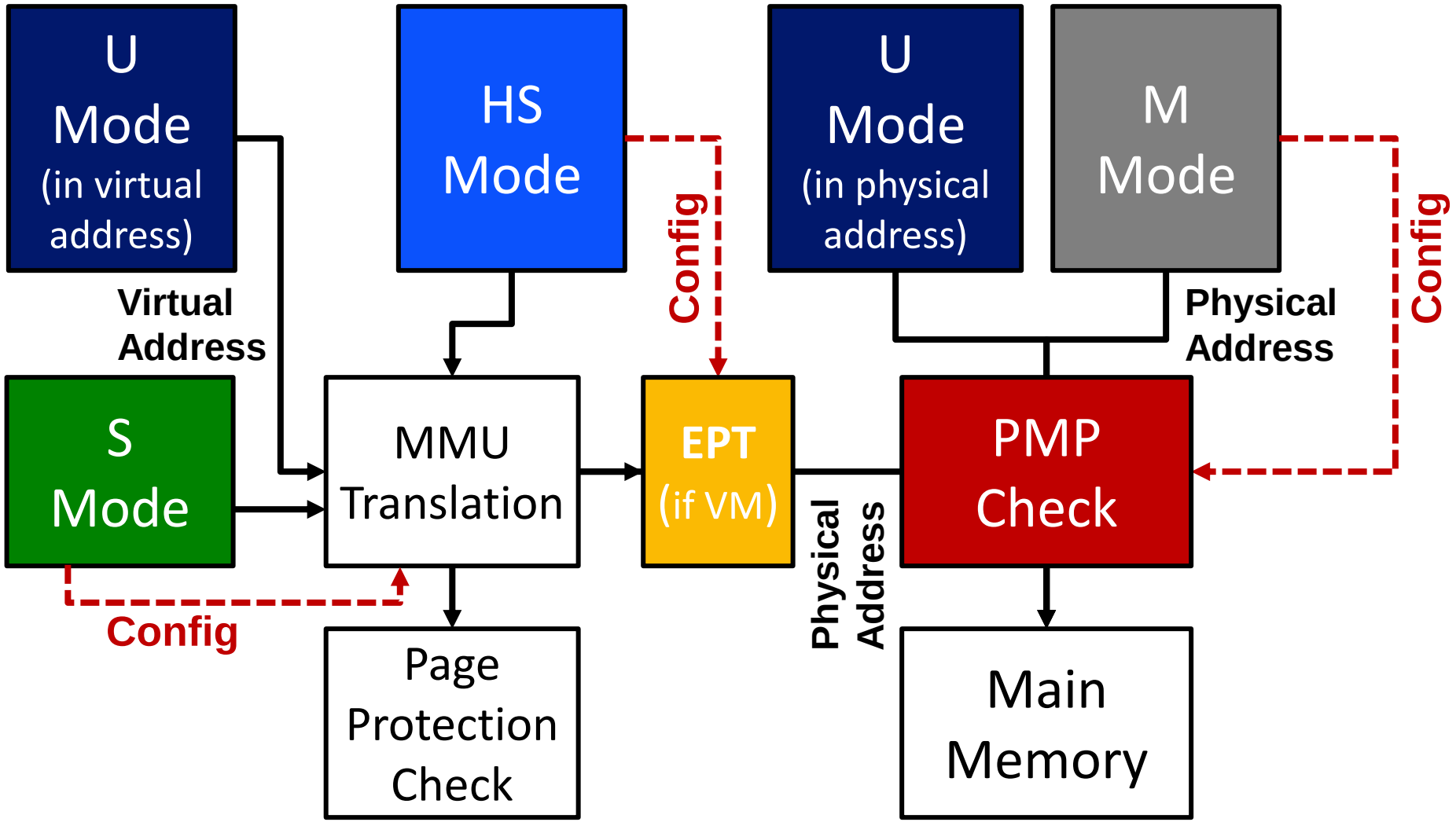
M-Mode Controls PMPs

- **M-mode** has access to entire machine after reset
- Configures **PMPs and ioPMPs** to contain each active context inside a physical partition
- Can even restrict M-mode access to regions until next reset
- M-mode can dynamically **swap PMP settings** to run different **security contexts** on a RISC-V hardware thread (*hart*)

Extended Page Tables (EPT) in HS Mode

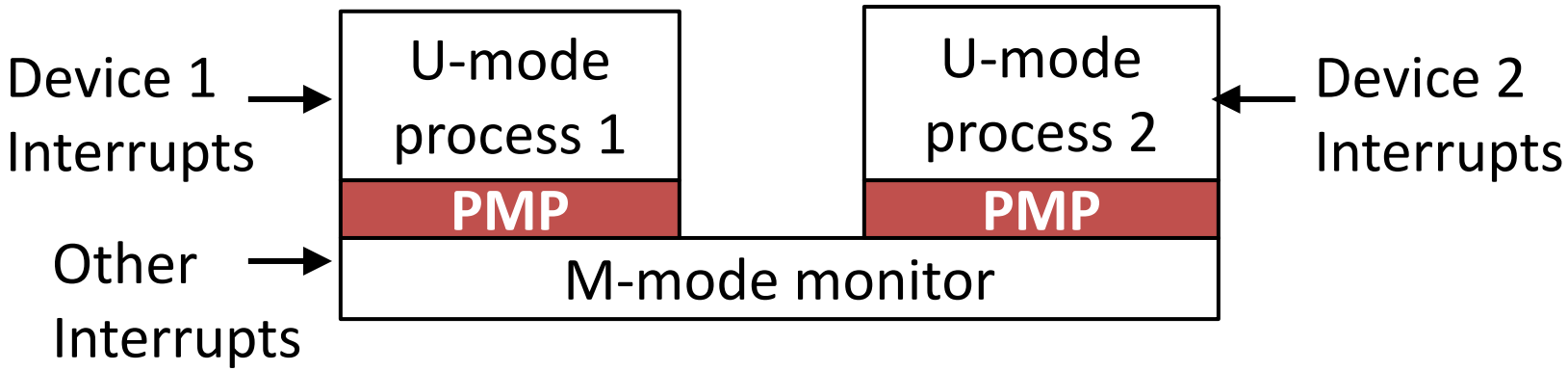


Memory Protection for RISC-V Modes



RISC-V Secure Embedded Systems (M, U modes)

- M-mode runs secure boot and runtime monitor
- Embedded code runs in U-mode
- Physical memory protection (PMP) on U-mode accesses
- Interrupt handling can be delegated to U-mode code
  - User-level interrupt support (N-extension)
- Provides arbitrary number of isolated security contexts

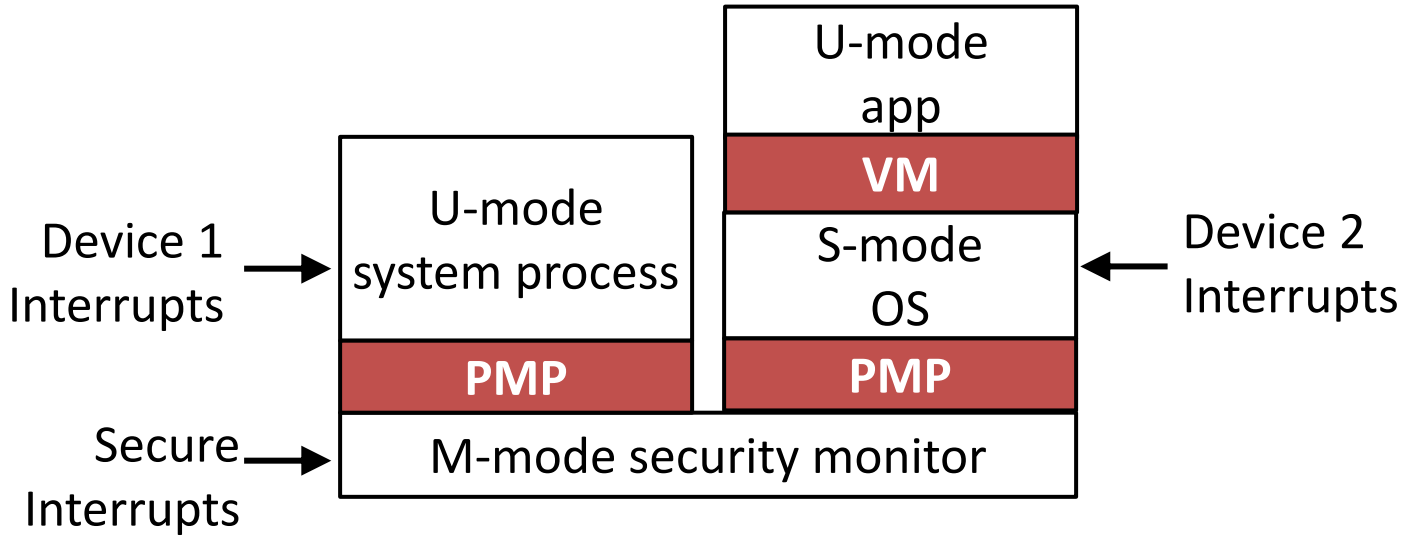


RISC-V Virtual Memory Architectures (M, S, U modes)

- Designed to support current Unix-style operating systems
- Sv32 (RV32)
  - Demand-paged 32-bit virtual-address spaces
  - 2-level page table
  - 4 KiB pages, 4 MiB megapages
- Sv39 (RV64)
  - Demand-paged 39-bit virtual-address spaces
  - 3-level page table
  - 4 KiB pages, 2 MiB megapages, 1 GiB gigapages
- Sv48, Sv57, Sv64 (RV64)
  - Sv39 + 1/2/3 more page-table levels

S-Mode runs on top of M-mode

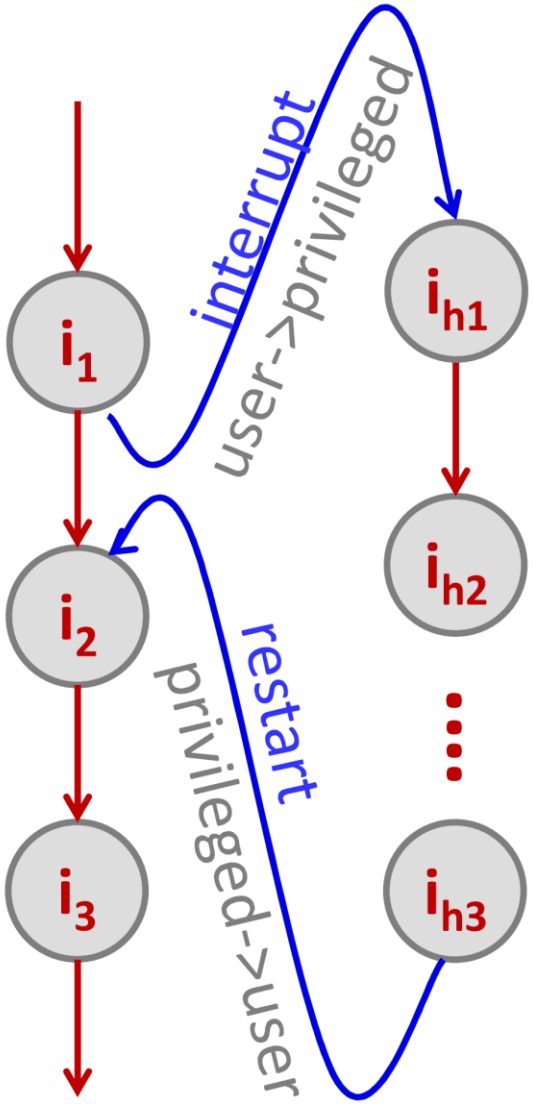
- M-mode runs secure boot and monitor
- S-mode runs OS
- U-mode runs application on top of OS or M-mode



- PMP checks are also applied to page-table accesses for virtual-address translation, for which the effective privilege mode is S. Optionally, PMP checks may additionally apply to M-mode accesses.
- PMP can *grant* permissions to S and U modes, which by default have none, and can *revoke* permissions from M-mode, which by default has full permissions.

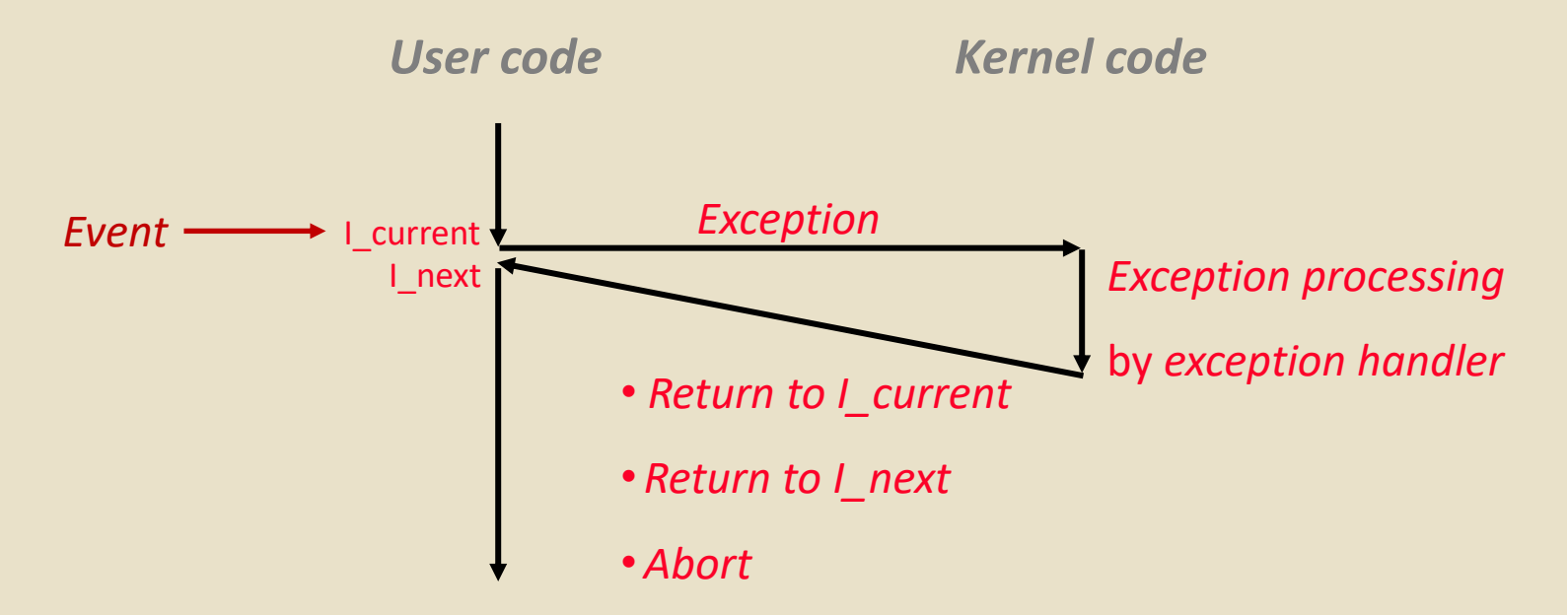
Privilege Level Change

- Combine privilege level change with **interrupt/exception** transfer
  – switch to next **higher privilege level** on interrupt/exception
  – get back to **lower privilege** level on return from interrupt/exception
- Interrupt/exception transfer is **the only gateway** to privileged mode
  – lower-level code can **never escape** into privileged mode
  – lower-level code **don't even need to know** there is a privileged mode

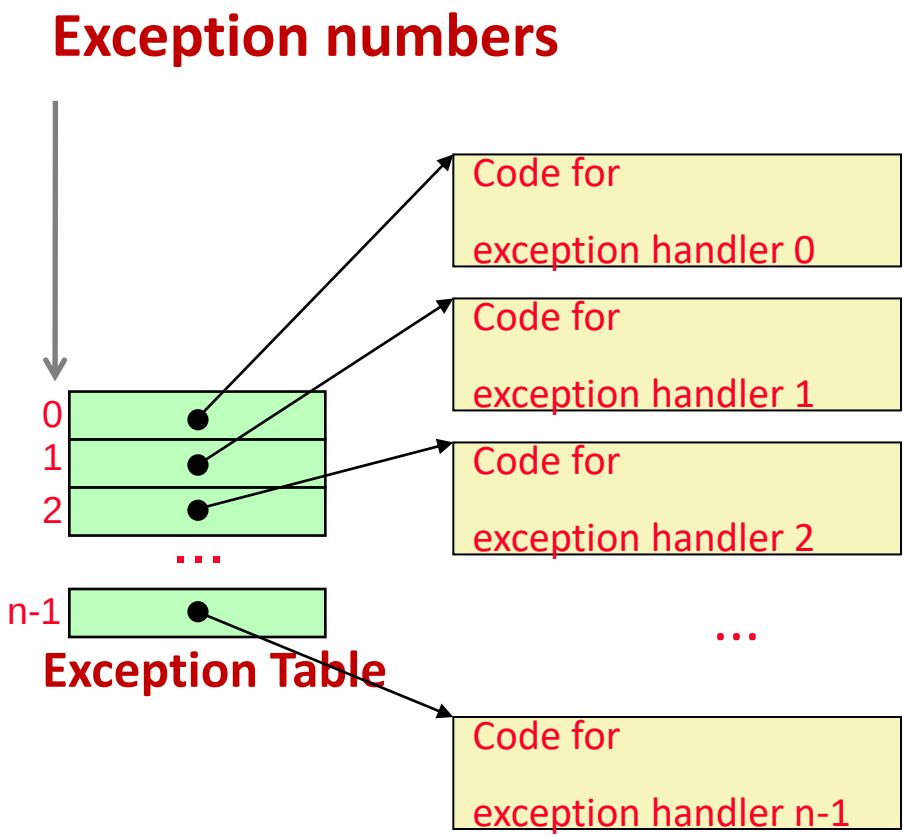


Exceptions

- An ***exception*** is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: divide by 0, arithmetic overflow, **page fault**, I/O request completes, typing Ctrl-C



Exception Tables



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to "**next**" instruction
- Examples:
  - Timer interrupt
    - Every few msec, an external timer chip triggers an interrupt
    - Used by the kernel to **take back control** from user program
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional, set program up to "trip the trap" and do something
    - Examples***: software interrupts***, ***system calls***, gdb breakpoints
    - Returns control to "**next**" instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: **page faults** (recoverable), **protection faults** (unrecoverable), floating point exceptions
    - Either re-executes the failed ("**current**") instruction or **aborts**
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: **illegal instruction**, parity error, machine check
    - **Aborts** current program

System Calls

- **Each x86-64 system call has a unique ID number**
- **Examples:**

Number	*Name*	*Description*
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

Virtual Memory Use Today - 1

- Servers/desktops/laptops/smartphones have full demand-paged virtual memory
  - Portability between machines with different memory sizes
  - Protection between multiple users or multiple tasks
  - Share small physical memory among active tasks
  - Simplifies implementation of some OS features
- Vector supercomputers have translation and protection but **rarely complete demand-paging**
- (Older Crays: base&bound, Japanese & Cray X1/X2: pages)
  - **Don't waste expensive CPU time** thrashing to disk (make jobs fit in memory)
  - Mostly run in batch mode (run set of jobs that fits in memory)
  - Difficult to implement **restartable vector instructions**

Virtual Memory Use Today - 2

- Most embedded processors and DSPs provide **physical addressing only**
  - Can't afford area/speed/power **budget** for virtual memory support
  - Often there is no secondary storage to swap to!
  - Programs custom written for particular memory configuration in product
  - Difficult to implement restartable instructions for exposed architectures
- Mostly **DNN accelerators** don't use (complex) virtual address
  - Working as IP or plug-in device, not running OS
  - Fixed task workloads and pattern
  - Still using some address mapping techniques (not complicated)
  - Hungry for energy and area, must be as efficient as possible

Next Lecture：Branch Prediction

Acknowledgements

- Some slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - David Patterson (UCB)
  - David Wentzlaff (Princeton University)

- MIT material derived from course 6.823
- UCB material derived from course CS252 and CS 61C